DSP 应用大观

TI DSP在视频传输和处理中的应用

张旭东　主编

電子工業出版社
Publishing House of Electronics Industry
北京 • BEIJING

内 容 简 介

本书主要介绍 TI DSP 在视频传输和处理中的一些应用实例，分四部分内容。第一部分介绍 DSP 和视频技术的一些基本概念，包括 DSP 系统开发的基本框架和视频编码基础；第二部分讨论 DSP 在 JPEG 和 MPEG 实现中的应用；第三部分是 H.264 在 DSP 上的实现，分别给出在 TMS320C6416 和 DM642 平台上的实现，包括 H.260 在 DSP 上实现的软件结构优化和算法效率优化；第四部分讲解 DSP 在图像增强方面的应用，这部分还介绍了从 MATLAB 算法到 DSP 实现的开发过程。本书可供视频传输和处理的专业技术人员阅读，也可作为相关专业大学高年级本科生和硕士研究生的参考资料。

图书在版编目（CIP）数据

TI DSP 在视频传输和处理中的应用 / 张旭东主编.—北京：电子工业出版社，2009.11

（DSP 应用大观）

ISBN 978-7-121-09782-9

Ⅰ. T … Ⅱ. 张 … Ⅲ. ① 数字信号－信息处理系统－应用－图像通信－数据传输 ② 数字信号－信息处理系统－应用－视频信号－信号处理 Ⅳ. TN919.8

中国版本图书馆 CIP 数据核字（2009）第 197080 号

责任编辑：万子芬（wzf@phei.com.cn）
印　　刷：
装　　订：
出版发行：电子工业出版社
　　北京市海淀区万寿路 173 信箱　邮编　100036
开　　本：787×1 092　1/16　印张：16.25　字数：416 千字
印　　次：2009 年 11 月第 1 次印刷
印　　数：4 000 册　　定价：35.00 元

凡所购买电子工业出版社图书有缺损问题，请向购买书店调换。若书店售缺，请与本社发行部联系，联系及邮购电话：（010）88254888。

质量投诉请发邮件至 zlts@phei.com.cn，盗版侵权举报请发邮件至 dbqq@phei.com.cn。

服务热线：（010）88258888。

《DSP 应用大观》丛书编委会

有关本丛书及其他与 DSP 相关选题的意见、建议和写作意向，请按以下方式联系：wzf@phei.com.cn　　010-88254461

丛 书 序 言

DSP 技术的发展与应用，正在我国教育界、科技界和工程界蓬勃地展开。数以百计的大学建设了 DSP 技术实验室，开设了相关的课程和实验；大量的相关教材、技术手册和应用书籍得到编写、编译和出版。更为重要的是，基于 DSP 技术的研究和开发，无论是涉及的范围，还是达到的深度，都令人叹为观止。以两年一度的 TI DSP 大赛为例，每次都有数十所大学的上百支代表队参赛，参赛者所表现出来的选题的广度、算法研究的深度，算法实现和系统设计及实现的娴熟程度，无不令人振奋。

随着教学、科研的发展和深入，教师、学生、以及科研和工程技术人员已经不再满足于对 DSP 的粗浅了解；市场的发育，对 DSP 技术的发展和应用也提出了越来越高的要求。在这样的形势下，编写和出版一套 DSP 应用汇编丛书，就成为一种强烈的需求，并迅速在出版社、TI 公司以及编写者之间达成了共识。

我们也注意到，在全球范围内，随着 DSP 技术应用范围的扩大和应用程度的深入，通用 DSP 器件的增幅在逐步减缓，而基于 DSP 核的各种 SoC、ASSP 以及嵌入式系统，正在以更快的速度发展。对于 DSP 工程师来说，开发算法并将算法在 DSP 芯片或 DSP 核上实现，还将仍然是长期的重要任务。本丛书的编写和出版，正是基于这样的认识和理解。

这套丛书是这样设计的：

按应用领域来分类，先在几个重要的领域，例如，通信信号处理、图像/视频信号处理、音频/语音信号处理、工业控制、通用信号处理算法、DSP 接口与软件工具等，各出一个选题。每个选题以 TI 网站上公开的 Application notes 为基本内容，为了便于读者理解和使用，各书的编译者对所介绍的内容，都不同程度增加了补充性的介绍。

这套丛书是开放的，这里所指的开放，包含以下两重意思：一方面，随着各领域的技术进步，新的算法和新的器件层出不穷，本丛书对新的算法及其实现的介绍也会继续下去；另一方面，欢迎广大的读者对丛书的选题和内容提出意见和建议，更欢迎有志者加入编写者的行列。

本丛书第一批选题的作者，是各高校多年从事 DSP 技术研究和实践的教师，以及他们的一些研究生，他们在各自的领域具有长期的知识积累和丰富的实践经验，为本丛书的选题、编写和出版付出了辛勤的劳动。

TI 公司对本丛书所使用的文档予以了授权，TI（中国）大学计划对丛书的编写和出版给予了一贯的支持和鼓励。电子工业出版社的编辑们，首先提出了本丛书的创意，积极参与了选题策划和论证，认真地完成了编辑和出版工作。在此，对所有为本丛书的选题、编写、出版作出贡献的单位和人士，致以深切的谢意和敬意。

希望这套丛书的出版，能对推动我国 DSP 技术的教育和应用起到微薄的作用，衷心希望得到广大读者的支持、意见和建议。

电子科技大学教授

2008 年 4 月

前　言

本书主要介绍 TI DSP 在视频传输和处理中的一些应用实例，可供工程技术人员和大学高年级学生参考。按照丛书主编彭启琮教授的设想，丛书根据不同的应用领域，以整理和编译 TI 的“Application Notes”为主，这个设想对于基本算法、语音处理等分册是非常适合的。的确，TI 的“Application Notes”中有大量的应用算法和系统的技术报告，但对于视频传输和应用领域，情况有些不同。在准备本书材料的时候，有关 H.264 这样最新的视频编码标准的技术报告很少，而且除了标准的概要介绍外，几乎没有实质性的内容，这是可以理解的，因为 H.264 在 TI DSP 上实现的编解码器有很高的商业价值，没有人会把这样的技术公开在网络资源中。鉴于这种情况，本书做了一些调整，对于 JPEG、MPEG 这些传统方法，我们整理了“Application Notes”中的相关技术报告，而对于 H.264 的相关技术，增加了我们实验室的相关学位论文和技术报告作为主要内容。

本书分为四部分。第一部分包括前两章，第 1 章概要，叙述 DSP 和视频技术的一些基本概念，并介绍 DSP 系统开发的基本框架；第 2 章给出视频编码技术的介绍，由于目前一线工程师大多缺乏视频编码的系统知识，本章概要性的介绍是有用的。

第二部分包括第 3 章、第 4 章，分别介绍 TI DSP 在 JPEG 和 MPEG 实现中的应用，这两章的体系是一样的，首先介绍标准的原理，然后编译了相关标准实现的技术报告。

第三部分由第 5 章、第 6 章构成，介绍 H.264 在 TI DSP 的实现，分别给出了在 TMS320C6416 和 DM642 平台上的实现。第 5 章是一篇完整的技术报告，包括 H.264 的详细介绍，H.264 在 DSP 上实现的软件结构优化和算法效率优化两个方面的内容，实际上，一个复杂算法在 DSP 上的实现都会存在软件优化和算法优化两个部分，缺一都很难达到好的性能，尽管本章介绍的是一个阶段性的研究结果，但其包含的内容对进入该领域的人很有参考价值；第 6 章则介绍了 H.264 在 TMS 320DM642 上的实现。

第四部分由最后两章构成，介绍 TI DSP 在图像增强方面的应用，在低照度的视频监控环境下，图像增强是很实用的工具，这部分还介绍了从 MATLAB 算法到 DSP 实现的开发过程。

本书是一本应用汇编性的图书，并不是一本完整的教材，本书实际是实验室集体工作的结果，本书主编只是将多人的工作汇编成册，并进行了整理和统稿，第 3 章、第 4 章的标准实现源自 TI 的“Application Notes”，第 5 章的主要内容来自魏振宇的硕士学位论文，第 6 章的主要内容来自范嘉略的技术报告，第 7 章、第 8 章的主要内容来自张众的技术报告，在此，谨向他们表示感谢，同时，感谢彭启琮教授的帮助和指导，感谢 TI 大学计划部沈洁经理长期对实验室的支持。

目　录

第 1 章　技术基础概要 …… （1）

1.1　数字视频编码标准的演进 …… （1）
1.2　数字视频编码国际标准概述 …… （3）
1.2.1　国际电信联盟（ITU-T）视频标准 H 系列 …… （3）
1.2.2　MPEG 系列视频标准 …… （4）
1.3　DSP 系统开发的基本流程 …… （6）
1.3.1　DSP 的发展及特点 …… （6）
1.3.2　DSP 系统的设计与开发 …… （8）
1.4　视频处理算法开发平台 …… （11）
1.4.1　DSP 程序开发的基本流程 …… （11）
1.4.2　DM642 开发平台 …… （15）
1.4.3　XDS560 JTAG 仿真器 …… （17）
1.4.4　DSP/BIOS 实时内核 …… （21）
1.4.5　CCS（Code Composer Studio） …… （25）
1.4.6　软/硬件接口 …… （26）
1.4.7　一个示例程序 …… （27）
参考文献 …… （31）

第 2 章　视频图像压缩编码基础 …… （32）

2.1　数字图像编码概述 …… （32）
2.2　图像的表示和编码质量的评价 …… （33）
2.2.1　静止图像格式 …… （33）
2.2.2　视频序列的常用格式 …… （35）
2.2.3　编码质量的评价 …… （37）
2.3　信息理论基础和熵编码 …… （38）
2.3.1　离散信源的熵表示 …… （38）
2.3.2　信源编码定理 …… （41）
2.3.3　Huffman 编码 …… （43）
2.3.4　算术编码 …… （45）
2.3.5　行程编码 …… （47）
2.3.6　有记忆信源的编码问题 …… （48）

2.4 量化 …… （49）
2.4.1 率失真函数 …… （49）
2.4.2 标量量化 …… （51）
2.5 预测编码 …… （55）
2.6 变换编码 …… （59）
2.6.1 一般图像变换 …… （59）
2.6.2 DCT 变换 …… （64）
2.6.3 变换编码 …… （66）
2.6.4 基于 HVS 的量化与码率分配 …… （68）
2.6.5 量化系数的扫描和表示方法 …… （70）
2.6.6 一个编码实例 …… （72）
2.7 块匹配运动估计与补偿 …… （73）
2.7.1 运动矢量的快速搜索算法 …… （75）
2.7.2 变块大小的分层运动估计 …… （79）
2.7.3 分数像素运动估计 …… （84）
2.7.4 重叠运动补偿预测（OMCP） …… （87）
2.7.5 双向预测 …… （88）
2.8 序列图像编码算法 …… （89）
2.9 各种图像压缩标准的应用目标和主要技术 …… （91）
参考文献 …… （94）

第 3 章 TMS320C6000 实现 JPEG 编解码器 …… （95）

3.1 JPEG 编码标准 …… （95）
3.1.1 JPEG 标准的工作模式 …… （95）
3.1.2 基本工作模式 …… （96）
3.1.3 其他工作模式 …… （101）
3.2 JPEG 在 C6000 上的实现 …… （104）
3.2.1 JPEG 编码器 …… （105）
3.2.2 JPEG 解码器 …… （111）
参考文献 …… （115）

第 4 章 MPEG 编码标准及其在 DSP 上的实现 …… （116）

4.1 MPEG-1 视频压缩标准 …… （116）
4.1.1 SIF 格式 …… （117）
4.1.2 MPEG-1 视频编码 …… （118）
4.1.3 MPEG-1 视频解码 …… （124）
4.1.4 MPEG-1 的其他问题 …… （125）
4.2 MPEG-2 …… （125）
4.2.1 MPEG-2 的运动估计 …… （126）

4.2.2 MPEG-2 的变换和扫描 …… （127）
4.2.3 MPEG-2 的可分级编码模式 …… （128）
4.2.4 MPEG-2 分档和分层 …… （129）
4.3 MPEG-4 …… （130）
4.3.1 MPEG-4 的组成 …… （130）
4.3.2 MPEG-4 视频编码原理 …… （133）
4.3.3 MPEG-4 中视频编码器的实现 …… （134）
4.3.4 MPEG-4 中的差错控制方法 …… （138）
4.3.5 MPEG-4 中的解码技术 …… （140）
4.4 基于 MS320C62x 的 MPEG-2 视频解码器实现 …… （141）
4.4.1 软件实现概述 …… （141）
4.4.2 算法描述 …… （142）
4.4.3 解码器的实现 …… （143）
4.4.4 与解码器的连接 …… （144）
4.4.5 程序的运行 …… （149）
参考文献 …… （150）

第 5 章 TMS320C6416 实现 H.264 …… （151）

5.1 H.264 概述 …… （151）
5.2 H.264 视频编解码器 …… （152）
5.3 H.264 的结构框架 …… （154）
5.3.1 H.264 的档和层 …… （154）
5.3.2 H.264 支持的视频格式 …… （156）
5.3.3 H.264 的码流格式 …… （156）
5.3.4 H.264 的帧结构 …… （157）
5.4 H.264 具体技术概述 …… （158）
5.4.1 帧内预测编码 …… （159）
5.4.2 运动估计 …… （160）
5.4.3 整数 DCT 变换 …… （164）
5.4.4 熵编码 …… （165）
5.5 实现 H.264 编解码的 TMS320C6416 平台 …… （167）
5.5.1 TMS320C6416 简介 …… （167）
5.5.2 CPU 的技术特点 …… （169）
5.5.3 NVDK（Network Video Development Kit）简介 …… （171）
5.6 H.264 在 NVDK 上的实现与优化 …… （175）
5.6.1 算法选择 …… （176）
5.6.2 编码器代码移植 …… （179）
5.6.3 代码优化 …… （180）
5.6.4 程序优化结果 …… （185）

5.7 算法优化 …… （185）
5.7.1 快速整像素运动估计算法——ARPS-4 …… （186）
5.7.2 基于早停止技术的亚像素运动估计快速算法 …… （188）
5.7.3 快速运动估计算法实验结果与分析 …… （190）
5.7.4 快速模式选择算法 …… （191）
参考文献 …… （198）

第 6 章 H.264 编码器在 TMS320DM642 上的实现和优化 …… （199）

6.1 TMS320DM642 EVM 介绍 …… （199）
6.1.1 DM642 的缓存结构 …… （200）
6.1.2 TMS320DM642 的视频接口 …… （201）
6.2 DSP 平台的程序开发问题 …… （202）
6.3 编码器实现 …… （203）
6.3.1 算法基本流程 …… （203）
6.3.2 代码移植 …… （204）
6.4 代码优化 …… （207）
6.4.1 项目级优化 …… （207）
6.4.2 指令级优化 …… （208）
6.4.3 缓存优化 …… （211）
6.4.4 优化结果 …… （214）
6.5 程序示例 …… （214）
参考文献 …… （221）

第 7 章 使用 CCS 开发视频图像增强算法 …… （222）

7.1 直方图均衡化的基本原理 …… （222）
7.2 实现代码 …… （224）
7.3 调试 …… （225）
7.3.1 DSP/BIOS 错误调试 …… （225）
7.3.2 使用 LOG 模块输出信息 …… （227）
7.4 算法性能优化 …… （228）
7.4.1 如何评估一个 DSP 算法的性能 …… （228）
7.4.2 程序优化 …… （231）
参考文献 …… （233）

第 8 章 使用 MATLAB 开发 DSP 的图像处理算法 …… （234）

8.1 MATLAB LINK FOR CODE COMPOSER STUDIO …… （234）
8.1.1 背景介绍 …… （234）
8.1.2 安装配置 …… （235）

8.2 示例程序 ····· （235）
8.3 使用 EMBEDDED MATLAB 构造 SIMULINK 模块 ····· （240）
8.3.1 Embedded MATLAB 简介 ····· （242）
8.3.2 如何使用 Embedded MATLAB 开发 Simulink Blocks ····· （243）
8.4 使用 MATLAB 开发的视频图像增强算法 ····· （244）
参考文献 ····· （246）

第 1 章　技术基础概要

随着通信技术和信号处理技术的发展，人们对多媒体信号的需求越来越多，要求的质量也越来越高，如何在现有的技术水平和硬件条件下，实现合理、优化、实时的多媒体通信终端设备和多媒体信息存储设备一直是近年来信号处理领域和相关产业界关注的话题。

我们知道，多媒体通信终端平台的实现主要有两点。第一，需要有快速、稳定的处理器作为多媒体处理的硬件平台；第二，需要有适合网络或者无线通信的视频通信协议，两者的结合才能产生高效的多媒体通信设备。多媒体存储设备则要求高质量的压缩编码和有效的检索、浏览功能。

多媒体信号处理主要针对音频、视频信号，本系列丛书中已有专门讨论音频处理的分册，所以本书重点介绍视频图像处理的实现技术。

目前，随着数字信号处理器（DSP）的高速发展，实现高效的视频图像处理有了可能，尤其是 TI（Texas Instruments）公司的 TMS320C64 系列产品及后续的 DM64X 系列，Davinci 系列和 OMAP 系列产品等，具有高主频、多流水线、高并行度及专用的视频信号处理指令和接口等优点，使其成为视频处理领域性能优异的 DSP 芯片类型之一。

针对当前网络带宽还不够，无线通信信道误码率较高的情况，稳定的低码率视频信号的需求一直都是必要的，而视频存储设备同样要求较高压缩率和方便的浏览功能。因此，各种视频压缩编码标准被制定并侧重于不同的应用领域，其中，H.264/AVC 是 ITU-T 视频编码专家组和 ISO/IEC 运动图像专家组联合提出的最新一代的视频编码标准。目前，在工程实际中，针对不同需要，有多种视频编码标准同时存在，可以根据需要选择合适的标准。

本书讨论用 DSP 实现数字视频的编码、传输和处理。本章首先概要介绍关于视频图像编码标准的发展，以及利用 DSP 实现视频处理的一些基本的流程，概要介绍 TI DSP 开发中的基本工具 CCS，并以 DM642 为参考平台，给出了一个视频图像采集和显示的例子。关于视频编码的关键技术在第 2 章给出更加详细的讨论，后续章节则介绍在几种不同 DSP 平台上实现典型视频处理算法的一些关键技术。关于各种 DSP 的 CPU 和指令集的详细叙述，请参考相关文档。

1.1　数字视频编码标准的演进

随着数字视频处理技术的快速发展，各种数字视频应用已经广泛深入到我们的日常生活中，如数字电视、视频电话、视频会议和视频监控，而第三代移动通信（3G）更是把无线视频传输作为其一大特点，这些快速发展的视频应用无疑推动了数字视频编码标准的制定。事实上，从 20 世纪 80 年代开始，ISO/IEC 和 ITU-T 这两大国际组织就已经不断推出一系列针对不同应用领域的数字视频编码标准，这其中包括 ISO/IEC 的 MPEG 系列和 ITU-T 的 H.26X

系列，图 1. 1 是数字视频编码标准的发展过程。

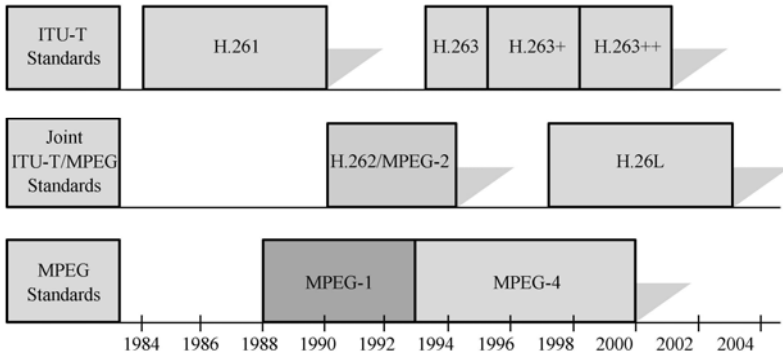


图 1.1　数字视频编码标准的发展过程

从图中可以看出，ITU-T 和 ISO/IEC 基本上是在相互独立的情况下开发各自的系列标准，其应用领域是不同的。MPEG 系列主要应用于数字娱乐（如 DVD、数字电视）或视频监控，而 H.26X 系列主要应用于实时的视频通信，包括视频电话、视频会议，等等。

虽然这些视频编码标准所采用的具体技术细节不同，但是它们都遵循运动估计、运动补偿、变换编码、熵编码这样的编码框架，之所以采用这样的模式，是为了最大限度地去除视频序列中的冗余信息，保持适度的复杂性。

通常，视频序列中主要包括三种冗余信息：空间冗余、时间冗余和统计冗余。

空间冗余是指视频信号中同一幅图像内相邻或相近像素之间具有的相关性。变换编码的目的就是去除这种冗余度。一般来说，所选择的变换应具有较好的能量集中特性，使得变换之后图像的能量集中在少数几个系数中。KLT 变换是目前能量集中性最好的一种变换，但是由于 KLT 变换的矩阵是不固定的，计算复杂度较大，因此，目前大多数视频编码器中采用的是 DCT 变换。与 KLT 变换相比，DCT 变换的能量集中性能非常接近，但是计算简便，有快速算法，因此得到了广泛的应用。空间冗余压缩技术也是静态图像编码（如 JPEG）采用的主要技术。

时间冗余是指视频信号中不同帧图像的像素之间具有的相关性，特别是在运动幅度比较小的序列中，这种相关性更加明显。运动补偿用来消除这种时间冗余度，它基于差分编码的思想，只对当前信号与参考信号之间的差值进行编码。即使一帧图像中原始像素的能量很高，经过运动补偿后得到的残差信号的能量也可以显著降低，有利于降低码率。

针对时间冗余的消除，目前最常采用的是运动估计和补偿技术（具体技术细节参考第 2 章），运动估计的目的就是为当前信号寻找最合适的参考信号。运动估计可以采用多种方式，包括基于块匹配的方法，基于光流的方法，等等。基于块匹配方法具有较好的准确性，并且运算量相对其他方法较低，因此在现代视频编码器中通常都采用这种方法。为了进行块匹配，一幅图像通常被分为若干个大小相同的单元，称为宏块（宏块的大小通常为 16×16），运动估计过程对每个宏块单独进行。

视频序列中的第三种冗余——统计冗余，是指码流中各符号间的相关性。为了消除这种冗余，一般采用熵编码的方法，最有名的熵编码方法是 Huffman 编码和算术编码，曾经被广泛地应用在各种图像和视频的编码算法中。后来人们对熵编码的算法进行了不断的完善，但

基本的思想并没有很大的变化。

除了上述三个主要部分，一般实用的编码器中通常还包含编码控制模块及码率控制模块，限于篇幅这里就不再赘述了。

1.2　数字视频编码国际标准概述

1.2.1　国际电信联盟（ITU-T）视频标准 H 系列

1. H.261

H.261 是最早出现的视频编码标准，在 1988—1993 年之间完成了协议的制定和修订。它首次采用了运动补偿预测编码加 DCT 变换的方式，奠定了以后视频编码技术的基础。其输出码率是 64 kb/s 的整数倍（1～31）。H.261 主要是针对 ISDN 的会议电视和可视电话等应用制定的，通过缓冲器控制产生恒定的输出码率。H.261 通过均衡图像质量和运动来优化带宽，所以，当图像快速运动时，质量会下降。

H.261 支持 CIF（352×288）和 QCIF（176×144）两种视频图像格式，采用简单的渐进扫描模式，帧率为 30 Hz。在 H.261 协议中，只允许使用 I 帧（Intra Frame）和 P 帧（Predict Frame）两种帧模式，运动估计采用整像素运动矢量，所以，在 H.261 标准中，压缩码率不是特别高。

2. H.263

H.263 是为了支持低速率通信而制定的标准，主要应用于 PSTN，ISDN 和无线网络，但也能够适应较大的动态范围，不限于低码率。对 CIF 图像，在 128kb/s～1Mb/s 码率范围内，H.263 可以获得比 MPEG-1 更好的压缩效果。

虽然 H.263 是从 H.261 基础上发展起来的，但是，它在低码率时，能够在不增加太多复杂度的情况下，提供更高的图像质量。此外，H.263 还包括 4 个可选模式，能够进一步提高压缩质量。

自从 1996 年 3 月 H.263 问世以来，就不断根据需求发展，先后出现了 H.263+，H.263++，H.26L 等后续标准。

1998 年，ITU-T 推出了 H.263 的第 2 版，叫做 H.263+。H.263+基于 H.263 标准，提出了一些新的可选特性以扩展 H.263 的应用领域并增强其编码效率。这些特性包括灵活视频格式、分级编码、补充增强信息（Supplemental Information）、增强 PB 帧编码及高级帧内编码方法等。其中分级编码包括空域分级、时域分级、SNR 分级模式，这几种模式的选择，增强了压缩信号的抗干扰能力，使得 H.263 码流适合于网络的传输。

H.263++是 H.263 的第 3 版，它在 H.263+的基础上，又增加了很多优化的选项，这些优化选项被整合后，作为必需的选项，构成了 H.264 的前身——H.26L。

H.263 系列标准具有里程碑的意义，在其中首次提出了很多概念，如变块大小的运动估计、初始运动矢量预测、无限制运动估计、多参考帧运动估计等都被其后很多标准所沿用。

3. H.264

H.264 是 ITU-T 和 ISO/IEC 联合制定的最新编码标准，它最先由 ITU-T 的 VCEG 于 1997 年提出，目标是提出一种更高性能（相对于当时的 H.263）的视频编码标准，其前身是由 H.263 发展起来的 H.26L，这里 L 代表 Long term，以区别 H.263 版本 2 和版本 3。由于其相对于 MPEG-4 的优良表现，2001 年年底，ISO/IEC 的 MPEG 加入到标准的开发中，与 VCEG 组成 JVT。到 2002 年年底，H.264 完成了所有技术工作，2003 年年底正式成为官方标准。该标准在 ITU-T 中被称为 H.264，而在 ISO/IEC 中成为 MPEG-4 的 Part 10（Advanced Video Coding Profile）。

目前，H.264 标准已经公布并被广泛采用，从各种实验来看，它达到了当初提出的目标，目前，很多 DSP 芯片具有标准分辨率下实现 H.264 编码器的能力和高清晰度下的实时解码能力。

1.2.2 MPEG 系列视频标准

1. MPEG-1

MPEG-1 是于 1991 年制定完成的。该标准主要是为了视频存储媒体（如 VCD）而制定的，目标应用码率为 1～1.5 Mb/s，提供 25 帧 CIF（352×288）VHS 质量的图像。MPEG-1 提供了视频序列的随机读取，快进和快退，视频序列的反向播放和压缩码流的可编辑性等功能。

MPEG-1 在典型的运动补偿预测编码（MCPC）框架基础上，应用了双向预测技术和半像素搜索，可以提供更好的编码质量和更高的压缩比。

双向预测技术允许编码帧参考前向和后向的参考帧，这样，MPEG-1 首次引入了 B 帧，B 帧不被用于其他 B 帧或 P 帧的运动补偿预测，因此，可以容忍更大的失真，并能提供相对于 I 帧和 P 帧更大的压缩比，大大提高了视频序列的压缩效率。

1/2 精度的运动估计，通过对整像素位置的亮度和色度进行插值，得到亚像素位置的信息，再进行运动估计。由于亚像素位置信息是对整像素的平滑，由此进行运动估计的结果会使得残差能量更低，压缩质量更好，但由此带来的运算复杂度也更高。

此外，MPEG-1 中还包含完整的音频编码、系统控制及一致性测试的规范。VCD 的广泛流行说明了 MPEG-1 的成功。MPEG-1 的图片组结构，使得其便于浏览，因此也被广泛应用于视频监控设备中。

2. MPEG-2

MPEG-2 是在 MEPG-1 基础上进一步发展起来的音视频编码标准，主要应用于广播级高质量音视频领域。目标码率为 3～35 Mb/s。该标准自 1990 年开始制定，于 1994 年完成标准化工作，目前被广泛应用于 SDTV，HDTV，DVD，DVB 等领域，在商业上取得了很大成功。在视频编码方面，与 MPEG-1 相比，MPEG-2 加入了两个主要的新技术：对隔行扫描的支持和可伸缩性编码。

对隔行扫描，在视频的每帧图像中，相邻行属于不同的场，一帧图像被分成两场：顶场和底场，也称奇场和偶场。在景物存在快速垂直运动时，相邻行的相关性会降低，影响编码效率。MPEG-2 通过支持场编码来解决这个问题。场编码也是电视信号采用的形式，MEPG-2 支持该项技术，也使之和电视视频兼容。

可伸缩性编码，顾名思义，就是根据信道质量或终端解码性能，对传输的码流进行实时调整，只传输或解码其中部分码流，仍可以得到完整的解码图像，只是相对于解码全部码流后的图像，或降低了分辨率，或降低了质量，或减少了帧率。MPEG-2 支持 4 种可分级编码模式：数据划分、SNR 分级、空间分级和时域分级，大大增强了该标准的实用性。

3. MPEG-4

MPEG-4 标准的制定是于 1994 年启动的，到 1999 年，第一个 MPEG-4 官方版本正式发表。与 H.263 系列相对应，MPEG-4 也能支持低于 64kb/s 的视频应用，同时它还能支持广播级的应用。它的制定目标是为了支持多种媒体的应用，特别是多媒体信息基于内容的检索和访问。

为了实现基于内容的视频交互及检索，MPEG-4 的一大特色是提出了视频对象的概念，并引入了基于对象的编码技术。MPEG-4 以 VO（Video Object）的概念来实现基于内容的表示。VO 的构成依赖于具体应用和系统实际所处的环境，在要求超低比特率的情况下，VO 可以是一个矩形帧，与传统的标准兼容。VO 也可以是场景中某一物体或某一层面，为画面中被分割出来的不同物体。每个 VO 由三类信息来描述：运动信息，形状信息和纹理信息。

此外，MPEG-4 还应用了很多技术，例如，多边形匹配、亚像素搜索、动态的块选择、重叠的运动估计、形状编码技术、sprite 技术及可扩展性编码，等等。

但由于实际存在对复杂视频中对象自动分割的困难，目前实时实现的 MPEG-4 编解码器并没有实现基于对象的编码技术。当前 MPEG-4 最流行也最成功的是其简单档（Simple Profile）和增强的简单档（Advanced Simple Profile）。前者基本和 H.263 类似，后者在 H.263 的基础上引入了 1/4 精度运动估计和全局运动估计技术，基于对象的概念和技术并未包括在这两档中。

4. MPEG-7 和 MPEG-21

MPEG 专家组还制定了两个多媒体技术标准，分别是 MPEG-7 和 MPEG-21，它们并不是视频压缩技术的标准，下面对它们进行简要介绍。

在 MPEG-4 之后，MPEG 专家组致力于 MPEG-7“多媒体内容描述接口”（Multimedia Content Description Interface）的制定。MPEG-7 可以形容为“基于语义的表示”。它指定一个用于描述各种多媒体信息的描述符标准集，还将定义其他描述符的方法标准化，也包括标准化描述符及其相互关系的结构。MPEG-7 标准化了一种描述语言，如 Description Definition Language（DDL）。视听材料（如静止图像，图形，3D 模型，音频，视频）和关于这些材料在一个多媒体表达中是如何结合等信息，通过 MPEG-7 的描述就能被索引和

搜索。

MPEG-21 是为了支持电子内容传输和电子商务而出现的国际标准，随着多媒体技术的迅猛发展，运营商为各自用户提供了丰富的信息和媒体业务，用户几乎可以随时随地享受这些服务。但是，对于不同网络之间用户的互通问题，至今仍没有成熟的解决方案。为了解决以上问题，MPEG 从 2000 年 6 月开始着手定义 21 世纪多媒体应用的标准化技术——MPEG-21 “Multimedia Framework”，MPEG-21 是一个可互操作和高度自动化的框架，而且这个框架还考虑到了数字版权管理 DRM（Digital Rights Management）的要求、对象化的多媒体接入及使用不同网络和终端进行传输等问题。

关于视频图像编码的基本方法和关键技术，在第 2 章有更详细的叙述。

1.3 DSP 系统开发的基本流程

DSP 是一种非常适合进行实时数字信号处理的微处理器，它具有可编程性好、可靠性高，灵活性大，易于大规模集成等优点，从诞生的第一天起就为业界所推崇。随着人们对实时信号处理要求的不断提高和大规模集成电路技术的迅速发展，DSP 芯片技术也不断推陈出新，发生着日新月异的变化。例如，传统的 DSP 是功能相对单纯的为信号处理结构优化设计的专用处理器，但近年来，DSP 和其他处理器技术的结合，出现了针对各种类型应用的真正的片上解决方案，例如，TI 的 DM642 是很好的视频解决方案，Davinci 和 OMAP 都是多核系统，集成了 ARM 和 DSP 多核处理器，可用于复杂的视频处理设备和移动设备。DSP 芯片在视频处理中的应用也越来越广泛。

1.3.1 DSP 的发展及特点

世界上第一款公认的 DSP 是于 1978 年诞生在 AMI 公司的 S2811，1979 年 Intel 公司推出的商用可编程器件 2920 是 DSP 芯片发展史上的重要里程碑。这两种芯片都没有现代 DSP 所拥有的单周期乘法器，但是已经是 DSP 芯片的雏形。1980 年日本 NEC 公司推出的 uPD7720 是第一款具有乘法器的商用 DSP 芯片。

在 DSP 领域最成功的企业应该是美国的德州仪器公司（Texas Instruments，TI），自 1982 年 TI 公司推出 TMS320 系列 DSP 中的第一款定点 DSP（TMS32010）以来，TI 的 DSP 已经发展了若干代，有 C1x，C2x，C2xx，C5x，C54x，C55x，C62x，C64x 等定点 DSP；有 C3x，C4x，C67x 等浮点型 DSP；此外还有 C8x 多处理器的 DSP，近年推出的针对多媒体处理的 DM64x，Davinci，以及针对移动终端的 OMAP 等。从最初的 16 位 DSP 发展到如今的 64 位 DSP，目前，处理能力最高的 C64x DSP 的最高主频已经达到 1.1GHz，处理能力为 8 800 MIPS。

除了 TI 公司以外，还有一些厂家的 DSP 产品在市场上占有一定份额。美国 AD（Analog Device）公司较著名的 DSP 芯片有 ADSP2101/2103/2105，ADSP2111/2115，ADSP2161/2162/2163/2164/2165/2166，ADSP2171/2173/ 2181 等定点 DSP，此外还有 ADSP/21000/21020，ADSP21060/21062 浮点 DSP。还有一些公司，如 AT&T，Motorola，NEC 公司的 DSP 芯片也各有特点。

DSP 主要的特点可以概括为以下几点。

1）采用哈佛结构

总线结构可以分为两种，一种是冯·诺伊曼结构，一种是哈佛结构。早期微处理器内部大都采用前者，其特点是程序和数据共用一个存储空间，程序数据总线共享，采用时分复用的方式共享总线。缺点是执行指令时只能串行执行，执行速度慢，吞吐量低。当高速运算时，不但不能同时取指令和操作数，而且会造成传输通道上的瓶颈。DSP 内部采用哈佛总线结构，数据存储空间和程序存储空间独立，数据总线和程序总线分开，这样就能够同时取数据（数据内存）和指令（程序内存），减少了冲突，大大提高了内存的访问速度。

TI 公司的 DSP 采用改进的哈佛结构，修正的地方有两点：一是数据空间和程序空间能够交换数据，有交叉数据通道；二是具有高速缓冲器，能够让 CPU 高速访问，大大节省读取指令和数据的时间，提高了运行速度。

2）流水线技术

所谓流水线操作，就是一个时钟周期同时进行多条指令的操作，取指令和执行指令同时进行，从而减少指令执行时间。DSP 执行一条指令，需要经过取指令、译码、取操作数和执行等几个阶段，每一阶段成为一级流水，当执行本条指令的时候，就可以同时进入下面指令的取指令、译码、取操作数的阶段。同时执行的指令个数称为流水深度，不同产品的流水深度不同，AD 公司的 ADSP 深度为 2 级，TI 公司的 C64x 和 Motorola 公司的 568xx 深度为 5 级。

3）多总线结构

许多 DSP 芯片内部都是多总线结构，这样就能够 1 个时钟周期内多次访问程序空间和数据空间。例如，TMS320C54x 内部有 4 条总线，可以在 1 个时钟周期内从程序存储器取 1 条指令，从数据存储器取 2 个操作数和向数据存储器写 1 个操作数，大大提高了 DSP 的运算能力。

4）特殊的指令系统

DSP 往往有一些功能强大的指令，这些指令都是专为数字信号处理领域的应用而设计的。例如，TMS320C64x 系列 DSP 的 LDDW 与 LDNDW 指令能够一次取 8 字节的数据。ADD4 一次能计算 4 组 8 位加法，此外，DSP 还有一些为专用领域设计的指令，如视频处理指令。C64xDSP 中的 SUBABS4 指令能够一次执行 4 组 8 位数据求差绝对值运算，DOTPU4 指令能够一次执行 4 组 8 位数之间的点乘运算，这些指令能够大大提高视频处理中运动估算法的运算速度。

5）多处理单元

DSP 内部一般都有多个处理单元，它们在一个时钟周期内能够同时进行运算，这样大大提高了 DSP 的并行处理能力。如 TI 公司的 C64x 系列 DSP 具有 8 个功能单元，为两组.S（Shift），.M（Multiply），.D（Data address），.L（Logic）单元。

6）采用硬件乘法器

一般早期计算机没有硬件乘法器，它们的算术逻辑单元只能完成加、减等运算，乘法和除法是通过移位等运算来完成的，因此，乘法在一般计算机中是非常耗时的运算。但是在信号处理中又有大量的乘法，DSP 芯片针对这些具体应用，从早期开始就设计了专门的硬件乘法器，如 C6000 系列 DSP 就有两个乘法单元，这也是 DSP 在主频不如普通 CPU，但是处理能力毫不逊色的一个重要原因。

7）寻址方式

在数字信号处理中需要用到大量的地址运算，在某些情况下，地址运算甚至超过了数据计算。数字信号处理器中都设计有一个特殊的硬件算术单元——地址产生器。地址的计算专门由硬件来完成，不需要额外的时间。目前一些微处理器也有独立的地址产生单元，处理一些特殊的地址运算，但是 DSP 的地址运算单元功能更强大，支持比特翻转寻址和循环寻址，可以大大加快傅里叶变换等运算速度，这些对于非哈佛结构的处理器可能需要更多的处理周期。

8）支持多处理器结构

单片 DSP 的处理能力还是比较有限，对一些运算能力要求很高，计算量很大的应用场合，单片 DSP 是无法胜任的。DSP 支持精细任务并发的紧耦合系统，可以将算法分配给多个 DSP，建立一个由链接支持实现的精细任务并发的多处理器阵列。

9）指令周期短

早期的 DSP 指令周期约为 400ns，运算能力约为 5MIPS。随着集成电路工艺的发展，DSP 的运行速度越来越快，目前 TI 公司处理能力最高的定点 DSP 主频已经达到 1.1GHz，运行速度达 8 800MIPS。

10）低功耗

作为一种嵌入式的处理器，和 PC 的 CPU 相比，DSP 具有体积小、功耗低的特点。一般为 0.5～4 W，采用低功耗技术的 DSP 只有 0.1 W，对嵌入式系统非常合适，而奔腾，powerPC 等处理器的功率为 20～50 W。

1.3.2　DSP 系统的设计与开发

随着 DSP 的不断复杂化及对 DSP 产品开发周期不断缩短的要求，设计调试 DSP 系统越来越依赖于 DSP 开发系统与调试工具。开发系统和调试工具方便设计者对软/硬件进行跟踪调试。

典型的 DSP 系统设计开发流程如图 1.2 所示。

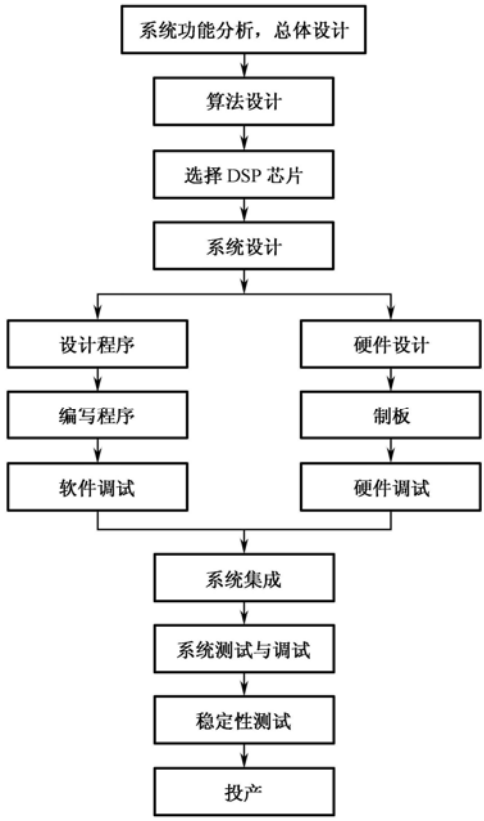


图 1.2　DSP 系统设计开发流程

（1）设计和开发 DSP 系统前，需要进行需求分析，设计者必须了解整个应用需求，包括信号处理与非信号处理的需求，通常可以采用数据流程图，数学运算式或自然语言描述。

（2）确定采用的算法，算法可以用高级语言或其他工具进行仿真，验证算法的可行性，从中进行算法选择，确定最佳方法。常用的仿真工具有 C 语言，MATLAB 和 SystemView 等。算法的选择要注意满足实际系统的要求，某些算法性能出众，但是在计算速度，内存消耗，计算精度等方面超过了实际系统计算能力的要求，这样的算法是不可取的。有些算法可能需要高昂的成本，超出了产品预算，这样的算法也是不能采用的。

（3）选择合适的 DSP 芯片，选择的标准要适合本系统的特点。如果系统有大量的数据运算，就需要高性能、计算能力强的 DSP 芯片，相应项目的成本也会随之增加。有些系统算法简单，数据运算量不大，可以选择运算能力相适应的 DSP 芯片，一方面节约成本，另一方面也不至于使 DSP 多余的计算能力浪费，达到最佳的性价比。

（4）设计实时的 DSP 系统。DSP 系统的设计包括硬件设计和软件设计两部分。设计者应该了解硬件设计和软件设计中的各种问题，并且在设计和开发之前对应用和可能出现的问题有清晰的了解。软/硬件设计相互依赖，软件设计者和硬件设计者之间要对彼此的工作有较好的了解，而不应该相互独立完成。如果有可能，可以软/硬件开发并行进行。

比较可靠的一种开发方式是采用一些第三方公司开发的 DSP 仿真板，如 EVM 板，DSK 板等，在上面进行软件开发与调试。这些开发板具有一些通用的硬件接口，用户只需要开发软件，从而大大缩短开发周期，而且可以有丰富的开发调试工具可以应用。

各 DSP 厂家推出了各种型号的 DSP，并提供了类似的软/硬件开发调试工具，大致有以下几种类型。

1）高级语言编译器

用 DSP 汇编语言编写程序，难度大，周期长，因此，通常 DSP 厂商都提供高级语言的设计方法，一般是 C 语言。如 TI 公司的 CCS（Code Composer Studio）就是这样一种功能强大的集成开发编译环境。用户可以使用标准 C 语言编写代码，此外，CCS 还有丰富的库函数可以调用，这部分函数执行效率都比较高。编译器将用户开发的代码编译成汇编语言。一般编译器虽然对代码有一定的优化，但是执行效率仍然比手工编写的代码差一些。重要的是，用户用高级语言设计代码，编写的难度降低了许多，如果需要提高代码执行效率，只需要将一些耗时模块抽取出来进行手动汇编改写就行了。

2）软件模拟器

这是一种脱离硬件的软件模拟仿真工具，它将代码加载后，可以在 PC 上模拟 DSP 的执行情况，例如，观察寄存器/存储器数值、单步操作、设置断点等。但是往往软件模拟速度很慢，而且无法模拟 DSP 与外设之间的操作，仅是对 DSP 芯片内部运行状况的模拟。对于 TI 的 DSP 来说，CCS 就有软件模拟器对其进行软件模拟。

3）硬件仿真器

仿真器是将 DSP 目标系统与调试平台连接起来的工具，能够真实地仿真程序在实际硬件环境下执行的情况，通过电缆将 PC 等调试平台和目标 DSP 系统连接，仿真器的工作界面和模拟器类似，但是有更多的调试功能。

通用 DSP 都有仿真接口，目前较新的 DSP 按照 JTAG 标准提供的 JTAG 仿真接口可以提供标准化的仿真功能。仿真器另一端通过并口或 USB 等端口和调试平台通信。TI 系列 DSP 的仿真器有 XDS510 系列、XDS560 等。

4）DSP 开发板

这种开发板大多是由 DSP 芯片制造商的合作厂家提供的一种包含 DSP 芯片、存储器、常用外围接口电路的电路板和相应软/硬件的系统仿真板。通常仿真板上具有仿真接口，可以通过仿真器和调试平台相连，也可以将电路板插在 PC 插槽中和 PC 通信。用户可以在仿真板上进行算法开发、验证、优化和调试，大大方便了基于 DSP 的软件开发。

表 1.1 列出了常用 DSP 系统开发工具。

表 1.1　常用 DSP 系统开发工具

开 发 内 容	软件开发工具	硬件开发工具
算法仿真	C 语言，MATLAB	PC
DSP 软件编程调试	DSP 软件编译器，模拟器等（如 CCS）	PC，DSP 仿真器，仿真板
DSP 硬件设计	EDA 软件	PC
DSP 硬件调试	相关支持软件	PC，仿真器，信号发生器，示波器，逻辑分析仪等
系统联合调试	相关支持软件	PC，仿真器，信号发生器，示波器，逻辑分析仪等

1.4　视频处理算法开发平台

这一节主要讲述如何使用 TI 公司所提供的软/硬件设施搭建视频处理算法的开发平台。本节分为以下 6 小节进行介绍。

第 1 小节综述 TI DSP 程序开发的基本流程；第 2 小节和第 3 小节介绍本节使用的硬件平台：DM642 评估模块与 XDS560 JTAG 仿真器；第 4 小节和第 5 小节分别介绍 DSP 开发中使用的软件工具：可扩展的 DSP/BIOS 内核与集成开发环境 CCS；最后一小节以一个视频采集-显示的示例代码展示如何实现软/硬件连接。

1.4.1　DSP 程序开发的基本流程

DSP 程序开发的基本流程如图 1.3 所示（以 TMS320C6000 系列为例），其中如果以 C 语言为开发工具，几个关键部分如下所述。

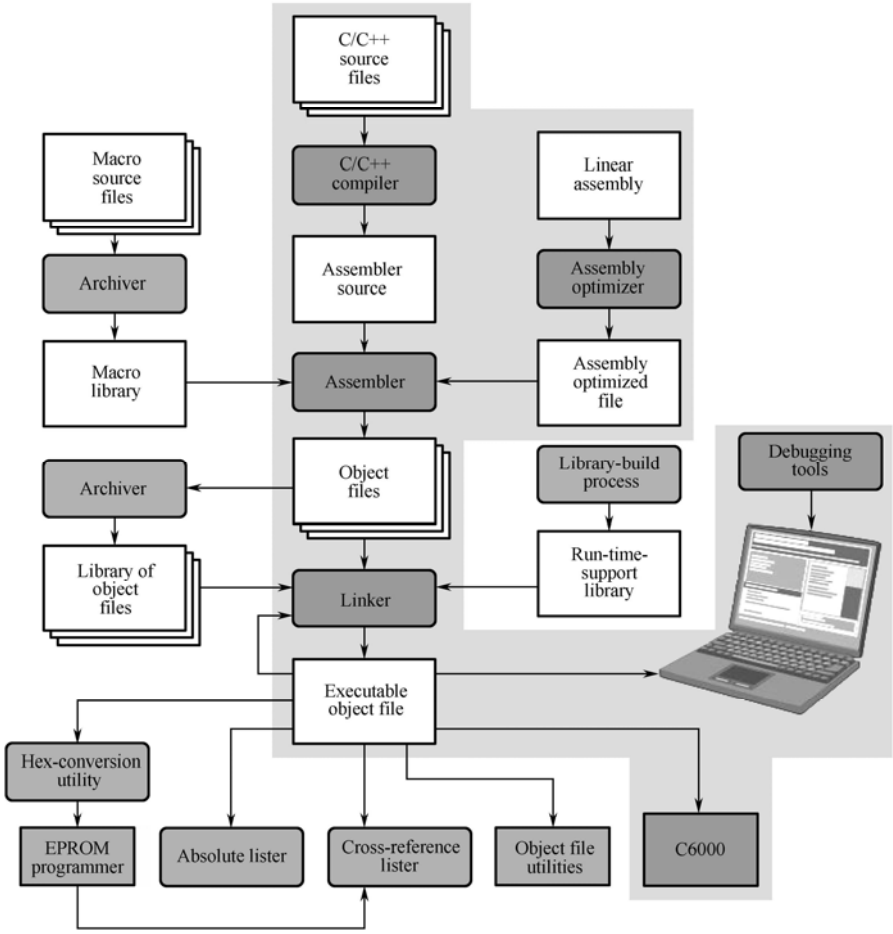


图 1.3　DSP 程序开发的基本流程

1）C/C++编译器（C/C++ Compiler）

输入为 C/C++源代码，输出为 C6000 的汇编代码。包括编译器，优化器和连接工具几个部分。因为视频处理采用的 TI DSP 大部分为 VLIW 结构，算法的效率在很大程度上决定于编译器的优化程度。

编译选项设置界面如图 1.4 所示。

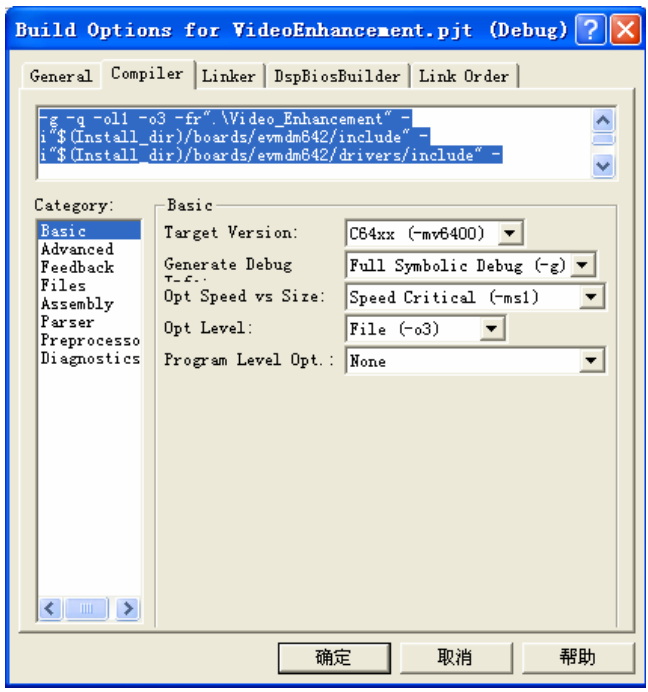


图 1.4　编译选项设置界面

2）汇编器（Assembler）

将汇编源文件翻译成机器语言目标文件，这里的机器语言使用 COFF（Common Object File Format）进行描述。

3）连接器（Linker）

连接目标文件以生成可执行文件（.out），它完成了以下任务：把各段分配到目标系统配置的存储器；对符号和段重定位，把它们分配到最终地址；处理引用问题等。

连接器选项设置界面如图 1.5 所示。

这几部分主要通过编译选项进行控制，CCS 提供了详细的图形化配置界面（见图 1.4、图 1.5），可以很直观地进行设置，但深入地了解各个编译选项的含义对程序调试和优化仍然相当重要，因此，下面的示例 1-1 以一个实际项目的编译设置为例，说明将 C 语言代码编译成在 DSP 上可执行文件的详细过程（该信息可通过项目文件夹下的.log 文件查看，这里因为篇幅所限仅保留部分内容）。

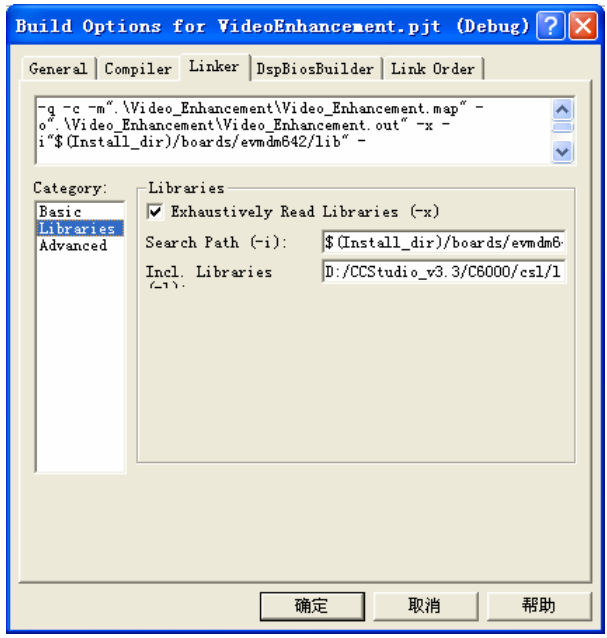


图 1.5　连接器选项设置界面

示例 1-1　编译设置。

```
        ----------------VideoEnhancement.pjt - Debug-----------------------
        [video.tcf] "D:\CCStudio_v3.3\bios_5_31_02\xdctools\tconf"
         -b     -Dconfig.importPath="D:/CCStudio_v3.3/bios_5_31_02/packages"
video.tcf
        [video_sd.c] "D:\CCStudio_v3.3\C6000\cgtools\bin\cl6x" -g -q -ol1 -o3
         -fr"./Video_Enhancement"
        -i"D:/CCStudio_v3.3/boards/evmdm642/include"
        -i"D:/CCStudio_v3.3/boards/evmdm642/drivers/include"
        -i"F:/Research/TIDSPBook/VideoEnhancement/VideoEnhancement"
        -i"D:/CCStudio_v3.3/C6000/csl/lib/include" -d"_DEBUG" -d"CHIP_DM642"
        -ml3 -ms1 -mv6400 -@"Debug.lkf" "video_sd.c"
        ……
        [Linking...] "D:\CCStudio_v3.3\C6000\cgtools\bin\cl6x" -@"Debug.lkf"
        <Linking>
        Build Complete,
          0 Errors, 0 Warnings, 0 Remarks.
```

下面来看一下上述编译中各项设置的具体含义。

video.tcf：DSP/BIOS 配置文件，通过 tconf 进行编译；

cgtools\bin\cl6x：CCS 的编译器；

video_sd.c：目标 C 源代码。

编译选项的含义如下所述。

-g：包含所有调试符号信息；

-q：抑制输出信息；

-ol1：告知编译器使用了库函数；

-o3: 对全文件进行优化；

-fr：设置目标文件目录；

-i：头文件搜索目录；

-d：预定义；

-ml3：改变 near 和 far 的设定；

-ms：控制代码尺寸；

-mv6400：优化目标为 C6400 系列；

-@filename: 将文件内容扩展为命令行的内容，可级联使用。在这里使用的“Debug.lkf”主要包含了所需链接的外部库信息，如示例 1-2 所示。

示例 1-2　连接器设置（Debug.lkf）。

```
-z -q -c -m"./Video_Enhancement/Video_Enhancement.map"
-o"./Video_Enhancement/Video_Enhancement.out" -x
-i"D:/CCStudio_v3.3/boards/evmdm642/lib"
-i"D:/CCStudio_v3.3/boards/evmdm642/drivers/lib"
-i"D:/CCStudio_v3.3/C6000/csl/lib"
-i"D:/CCStudio_v3.3/C6000/xdais/lib"
-i"D:/CCStudio_v3.3/bios_5_31_02/packages/ti/bios/lib"
-i"D:/CCStudio_v3.3/bios_5_31_02/packages/ti/rtdx/lib/c6000"
-i"D:/CCStudio_v3.3/C6000/cgtools/lib"
-l"D:/CCStudio_v3.3/C6000/csl/lib/cslDM642.lib"
"F:\Research\TIDSPBook\VideoEnhancement\VideoEnhancement\video.cmd"
```

Debug.lkf 中各设置参数含义如下所述。

-z：连接指定的目标文件；

-c：运行时自动初始化变量；

-m：指定输出的 map 文件；

-o：输出.out 文件名；

-x：强迫重新读取库文件；

-i：搜索路径；

-l：链接库名称。

最后一行的*.cmd 文件为项目自身的控制文件，描述了需要链接库的顺序和对应的内存配置，其中 video.cmd 的内容如示例 1-3 所示。

示例 1-3　CMD 文件。

```
/* include config-generated link command file */
-l videocfg.cmd
/* include library for the IOM video driver */
-l vport.l64
/* include library for the EVMDM642 board support library (BSL) */
-l evmdm642bsl.lib
```

其中 videocfg.cmd 由 DSP/BIOS 的 tconf 工具编译生成，通过 video.tcf 文件配置存储映射关系和模块组成。后面的两个语句则包含了两个针对 DM642 开发板的库文件：vport.l64 和 evemdm642bsl.lib。在 DSP 开发中有几个重要的库文件将会经常遇到，下面进行总结。

CSL：Chip Support Library（CSL） 提供了一组可用于控制芯片外设的 API，例如，$CCSInstallDir\C6000\lib\cslDM642.lib 提供对 DM642 的支持。

BSL：Board Support Library（BSL）提供一个能够控制开发板上设施的 C 语言接口（API），例如，$CCSInstallDir\boards\evmdm642\lib\evmDM642bsl.lib。

RTS：Run-Time Support Library（RTS）定义一组运行时支持函数来完成如 I/O、字符串操作、三角函数等任务，如$CCSInstallDir\C6000\cgtools\lib\ rts6400.lib 提供对 C6 400 系列运行时的支持。

1.4.2　DM642 开发平台

TMS320DM642 是 TI 公司 2003 年推出的一款针对多媒体处理领域应用的高性能 32 位定点 DSP，基于 C64x 核心架构，集成了丰富的外围设备和接口，主频最高可达 720 MHz，并行处理指令的能力最大可达每个指令周期处理 8 条 32 位指令，因此最大指令处理速度为 5760MIPS。该 DSP 为 548 脚 BGA 封装，集成化程度很高。其主要的外围设备接口包括：

（1）3 个可配置的视频接口（VP0，VP1，VP2），可以和视频输入、输出或传输流输入无缝连接；

（2）VCXO 控制端口（VIC）；

（3）10/100 Mb/s 以太网口（EMAC）；

（4）数据管理输入、输出模块（MDIO）；

（5）多通道音频串行端口（McASP）；

（6）I^2C 总线模块；

（7）2 个多通道有缓存的串行口（McBSP）；

（8）3 个 32 位通用定时器（Timer）；

（9）可配置的 16 位或 32 位主机接口（HPI16/HPI32）；

（10）66 MHz 32 位 PCI 接口；

（11）通用 I/O 端口（GPIO）；

（12）64 位外部存储单元接口，支持和同步或异步存储单元的连接。

图 1.6 为 DM642 和外设连接的结构框图。对于 DM642 的体系结构，需要特别注意 DSP 核是通过 EDMA 控制器（Enhanced DMA Controller）与外围设备进行连接，这一方面为程序优化提供了更大的余地，同时在程序设计时也要考虑数据一致性的问题，具体的技术细节可见 TI 的文档《TMS320C6000 DSP Enhanced Direct Memory Access Controller Reference Guide》（SPRU234）。幸运的是我们在编写基于 DSP/BIOS 的程序时并不用对 DMA 实现的细节全盘了解，在 DSP/BIOS 中提供有 DAT 模块，以完成常用的 DMA 操作，如 DAT_open（打开 DMA 通道），DAT_copy（DMA 数据传输）等。

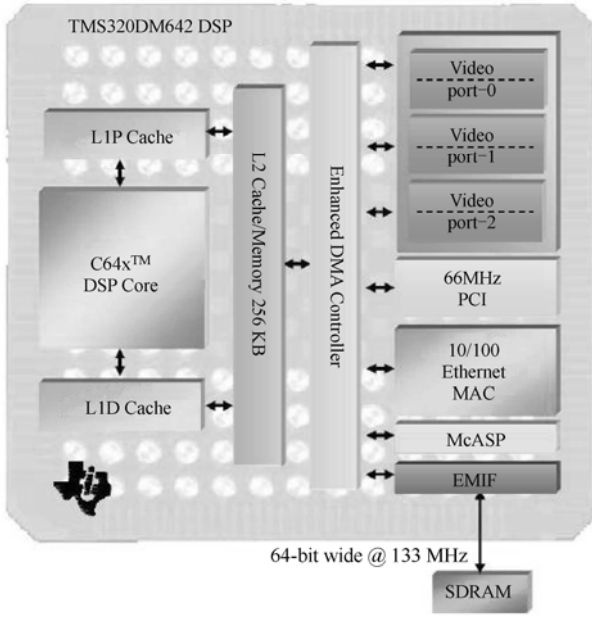


图 1.6　DM642 和外设连接的结构框图

图 1.7 为 DM642 实际开发板的照片，其中各功能模块已在图中标识。

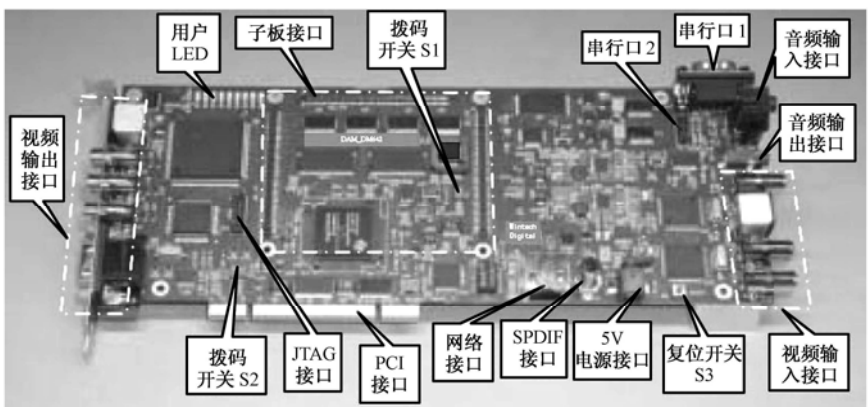


图 1.7　DM642 开发板

对应地，图 1.8 为 DM642 EVM 各外设之间的连接关系图，这里重点介绍两个与视频处理相关的模块：视频解码模块和视频编码模块。

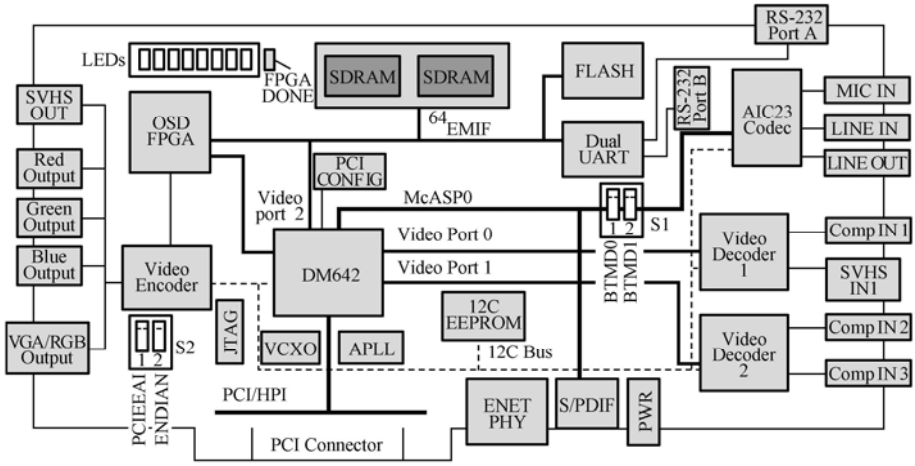


图 1.8　DM642 EVM 各外设之间的连接关系

视频解码模块：将模拟摄像头捕捉的模拟信号转化为数字信号，常用在 DM642 EVM 上的为 TVP5146 系列（单芯片）。

视频编码模块：将数字信号转化为合适格式以提供输出，如复合视频输出、VGA 视频输出及 HDTV 高清视频信号的输出等，常用在 DM642 EVM 上的为 SAA7105 系列（单芯片）。

1.4.3　XDS560 JTAG 仿真器

仿真器的目的在于将目标电路板连接至主机平台，并在主机和目标处理器之间提供数据传输机制。TI 在十多年前发明了基于 JTAG（Joint Test Action Group，IEEE 1149.1 standard）扫描的仿真，自此以后，该方法一直为嵌入式系统开发广为采纳。与较早且更昂贵的“内电路仿真”或“ICE”技术相比而言，JTAG 仿真更受开发者的青睐。内电路仿真的基本原理是在电路中内置仿真实现，将目标处理器替换为不同的器件，该器件“扮演”原始器件，其附加的引脚使器件上的内部结构（如总线）可见。 电路内置仿真具有很大的局限性，导致支持 200 MHz 以上的高速处理器所需要的成本让人望而却步。JTAG 仿真技术则省去了这些调试成本和困难，方法是直接与处理器进行通信，从而彻底避免专用的仿真器件。标准 JTAG 连接器可让数据以非插入方式在片上和片外移动，而不会中断正在执行的器件。然后，TI 在 DSP 上使用附加的仿真逻辑对此功能进行了补充，旨在提供更高的可视性并提供对寄存器和其他内部功能（如片上高速缓冲存储器）的访问。

JTAG 仿真器的引脚接口并不存在一个官方的标准，目前比较常见的有 ARM 2×7 pin，2×10 pin / MIPS EJTAG（2×7 pin）等多种连接方式，而 DM642 上的 JTAG 口的引脚为 2×7 pin，各引脚定义如图 1.9 所示。

TMS	1	2	TRST-
TDI	3	4	GND
PD（+3.3V）	5	6	no pin (key)
TDO	7	8	GND
TCK-RET	9	10	GND
TCK	11	12	GND
EMU0	13	14	EMU1

图 1.9　DM642 上 JTAG 接口引脚定义

值得注意的是 TI 推出的另一套多媒体开发平台 Davinci 上的 JTAG 接口为 2×10 pin，与现有的仿真器无法通用，需要另外购买转接头才可使用。

XDS560 仿真器是 TI 公司开发的新一代 JTAG 仿真器（见图 1.10，图 1.11），与前代产品 XDS510 系列相比，它具有更高的实时数据传输速率与代码下载速度。同时， XDS560 还提供了两种对于视频处理算法调试相当重要的技术：实时数据交换（RTDX）和高级事件触发（AET），现简要介绍如下。

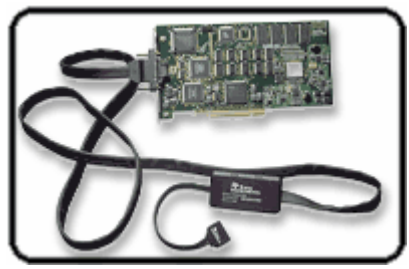


图 1.10　XDS560 仿真器

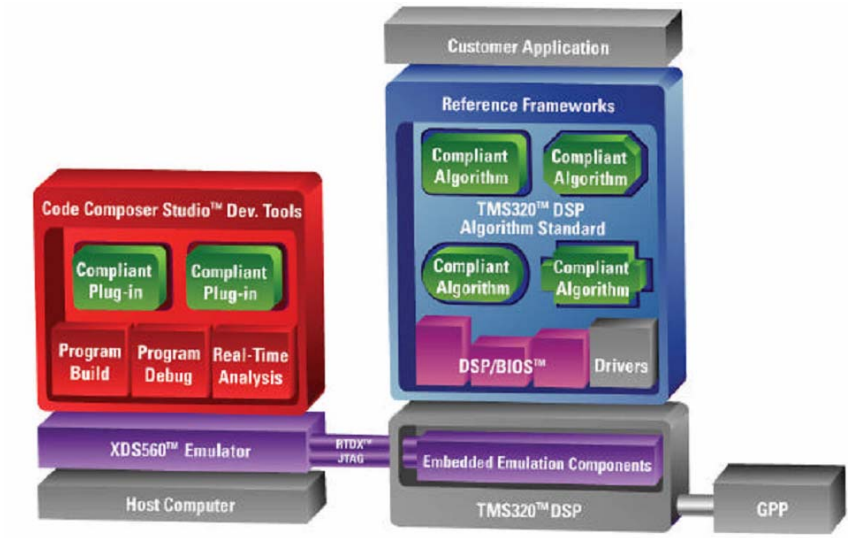


图 1.11　XDS560 的系统框图

1. 实时数据交换（Real-Time Data Exchange，RTDX）

过去最常用的硬件调试方式为使用断点来暂停应用程序，然后再与主计算机交换数据“快照”，从而获得当前程序的运行信息，这种技术称为“停止模式调试”。这种插入式方法会误导开发者，因为中止的高速应用程序的隔离快照不能准确显示系统的实际情况。为解决这个问题，TI 开发了实时数据交换技术，RTDX 的主要目的在于使设计人员能持续实时看到应用程序的直观视图。从实现角度来看，RTDX 能在目标和主机之间进行实时异步数据交换，而不会停止目标应用程序。我们也可以把 RTDX 数据链路看成在 DSP 应用和主机之间提供一个“数据管道”。这种双向能力允许开发者访问应用程序的数据以获取当前 DSP 信息的直观视图，或者模拟给 DSP 输入数据（在实际传感器硬件可用之前），这些为开发者有效地缩短了开发时间。

除了速度、实时数据传输等优势外，RTDX 数据交换的另外一个优点在于可以通过行业标准的 Microsoft 组件对象模型（COM）API 来查看，从而允许数据流被其他符合 COM 的应用程序（如 Microsoft Excel、Visual Basic）或信号处理和获取软件包（如 MathWorks 的 MATLAB，或美国国家仪器公司的 LabVIEW）进行处理。一个范例是 MATLAB 在 Target for Ti C6000 产品包中所提供的 Video Surveilance 示例程序，如图 1.12 所示。而具体的 MATLAB 与 DM642 的联合开发将在第 8 章专门讲述。

有两种类型的 RTDX。XDS510 类仿真器支持现有形式的 RTDX，它称为“标准”RTDX，能够处理 10 KB/s 和更高的数据速率。另外一种则为高速 RTDX，XDS560 可以向启用的处理器的高速 RTDX 提供 2 MB/s 以上的带宽。在处理器上不具有高速 RTDX 特性的条件下，XDS560 也可兼容标准 RTDX，并可把速度升高到 130 KB/s。

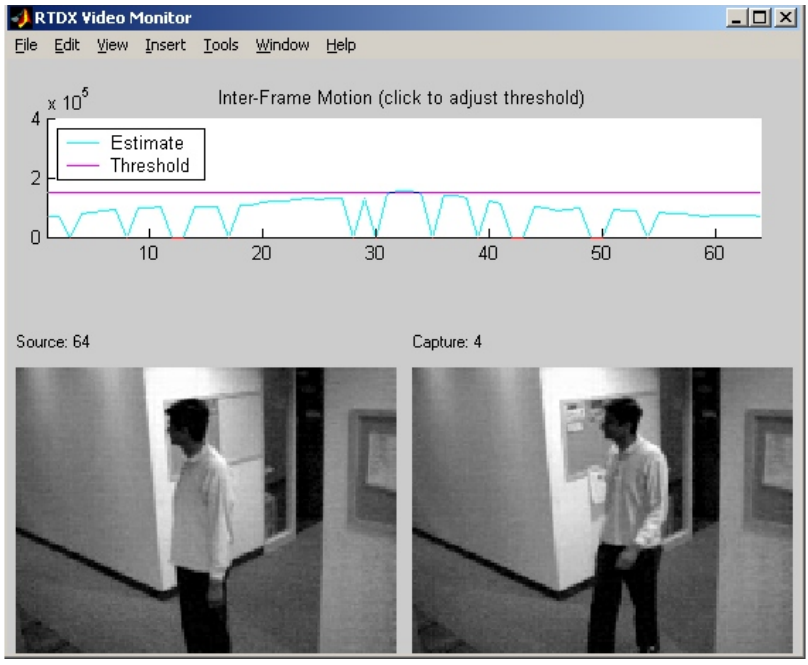


图 1.12　Demo：MATLAB 调用 RTDX

2. 高级事件触发（Advanced Event Triggering，AET）

高级事件触发是另外一种相当重要的调试技术，它提供了一种检测目标处理器事件组合，再执行相关调试操作的功能。简而言之，高级事件触发可以实现。

（1）直接在源码窗口中执行最频繁需要的调试任务，如硬件断点和监视点（见图 1.13）。

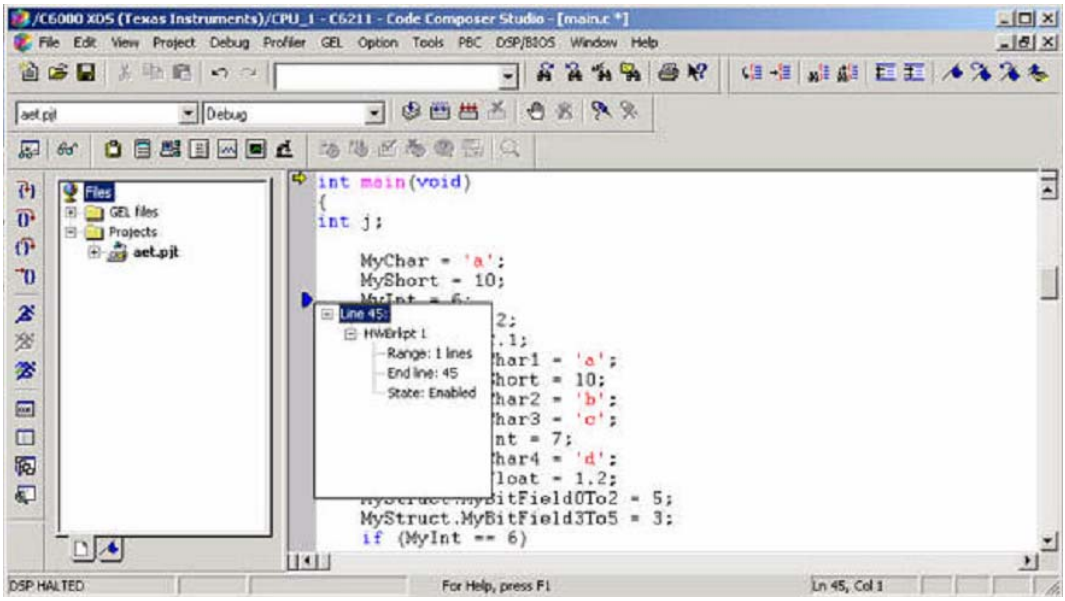


图 1.13　AET：源码窗口监控

（2）通过事件分析窗口管理所有调试任务（见图 1.14）。

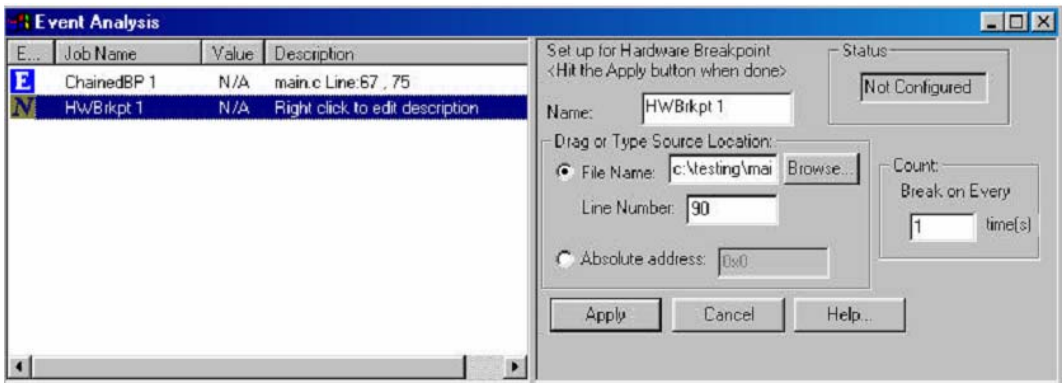


图 1.14　AET：事件分析窗口

（3）通过设置复杂组合和序列的检测事件查找故障（见图 1.15）。

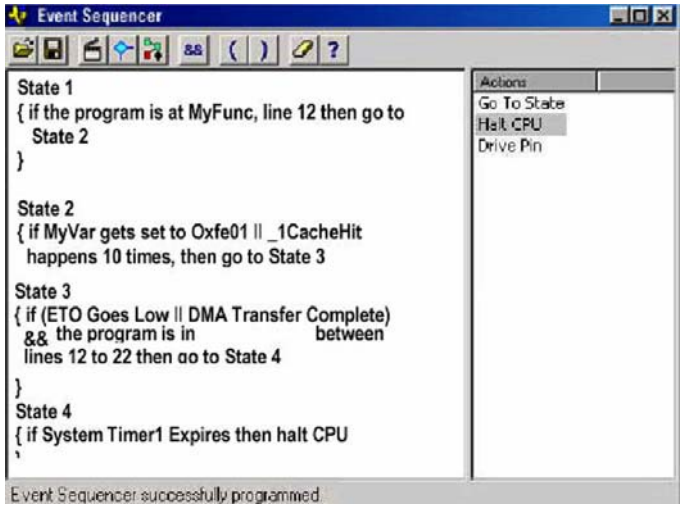


图 1.15　AET：事件序列窗口

熟练使用以上特性可以大大地提高在 DSP 上调试程序的能力，具体的技术细节可以参考 TI 文档《XDS560 Emulation Technology Brings Real-Time Debugging Visibility to Next-Generation High-Speed Systems》（SPRA823）或参考文献[1]。

1.4.4　DSP/BIOS 实时内核

DSP/BIOS 是一个功能丰富、可扩展的内核服务集，开发人员可以用来管理系统级的资源和构建 DSP 应用的基础架构。目标应用程序通过在源程序中嵌入相应的 API 调用唤醒 DSP/BIOS 的运行服务。这些服务除了分析调试进程和外设资源配置之外，还包括以下内核模块。

硬件中断：提供了从硬件中断至 DSP/BIOS 内核的接口；

软件中断：使用程序堆栈无法放弃的轻量级抢占式线程；

任务管理：可中断的独立线程；

周期功能：时间触发的轻量级线程；

信箱：任务间的同步数据交换机制；

信标：统计信标；

时钟：与硬件定时器的接口；

流：任务的流输入、输出；

管道：软件中断的流输入、输出；

内存管理器：低开销动态内存分配。

DSP/BIOS 在 DSP 开发系统中的作用见图 1.16。

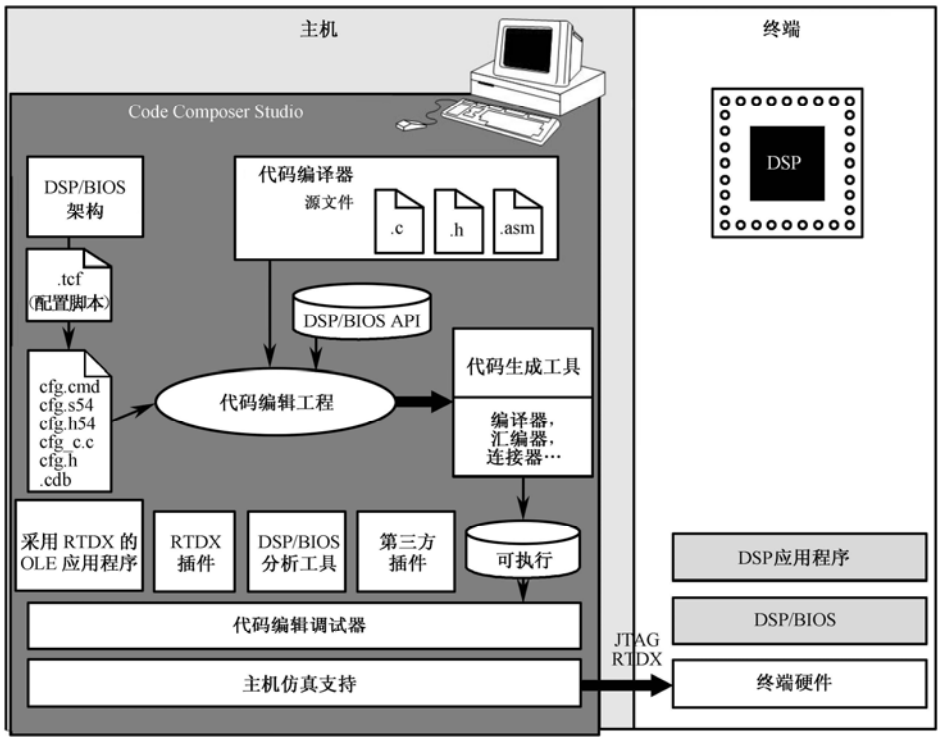


图 1.16　DSP/BIOS 在 DSP 开发系统中的作用

DSP/BIOS 的另外一个重要特点是其易配置性。DSP/BIOS 本身是可裁减的，它只把直接或间接调用的模块和 API 连接到目标文件中，这些都是静态的，通过 tcf（旧版本的 DSP/BIOS 通过 cdb 文件）进行配置。DSP/BIOS 提供了 tconf（Textual Configuration）工具集，将以上文本描述的配置转化为库文件和内存描述文件（见示例 1-4 编译设置）。一个配置文件示例（片段）如下所示。

示例 1-4　TCF 配置文件。

```
%001 environment["ti.bios.oldMemoryNames"] = true;
%002
%003 /* loading the generic platform */
%004 var params = {};
%005 params.clockRate = 720.000000;
%006 params.deviceName = "DM642";
%007 params.catalogName = "ti.catalog.c6000";
%008 params.regs = {};
%009 params.regs.l2Mode = "4-way cache (0k)";
%010 utils.loadPlatform("ti.platforms.generic", params);
%011
%012 /* enabling DSP/BIOS components */
%013 bios.GBL.ENABLEINST = true;
%014 bios.MEM.NOMEMORYHEAPS = false;
%015 bios.RTDX.ENABLERTDX = true;
```

```
%016 bios.HST.HOSTLINKTYPE = "RTDX";
%017 ……
%018
%019 /* applying user changes */
%020 bios.SDRAM = bios.MEM.create("SDRAM");
%021 bios.SDRAM.comment = "This object defines space for the DSP's off-chip
 memory";
%022 bios.SDRAM.base = 0x80000000;
%023 bios.SDRAM.len = 0x2000000;
%024 ……
%025 bios.tskLoopback = bios.TSK.create("tskLoopback");
%026 bios.trace = bios.LOG.create("trace");
%027 bios.VP0CAPTURE = bios.UDEV.create("VP0CAPTURE");
%028 bios.VP1CAPTURE = bios.UDEV.create("VP1CAPTURE");
%029 bios.VP2DISPLAY = bios.UDEV.create("VP2DISPLAY");
%030 ……
%031 bios.LOG_system.bufSeg = prog.get("SDRAM");
%032 bios.LOG_system.bufLen = 0x400;
%033 ……
%034 bios.TSK.STACKSEG = prog.get("SDRAM");
%035 bios.tskLoopback.comment = "Loopback Task";
%036 bios.tskLoopback.fxn = prog.extern("tskVideoLoopback");
%037 bios.tskLoopback.stackMemSeg = prog.get("SDRAM");
%038 ……
%039 bios.VP0CAPTURE.fxnTable = prog.extern("VPORTCAP_Fxns");
%040 bios.VP0CAPTURE.fxnTableType = "IOM_Fxns";
%041 bios.VP0CAPTURE.params = prog.extern("EVMDM642_vCapParamsPort");
%042 bios.VP1CAPTURE.fxnTable = prog.extern("VPORTCAP_Fxns");
%043 bios.VP1CAPTURE.fxnTableType = "IOM_Fxns";
%044 bios.VP1CAPTURE.deviceId = 1;
%045 bios.VP1CAPTURE.params = prog.extern("EVMDM642_vCapParamsPort");
%046 bios.VP2DISPLAY.fxnTable = prog.extern("VPORTDIS_Fxns");
%047 bios.VP2DISPLAY.fxnTableType = "IOM_Fxns";
%048 bios.VP2DISPLAY.deviceId = 0x2;
%049 bios.VP2DISPLAY.params = prog.extern("EVMDM642_vDisParamsPort");
%050 ……
%051 bios.GBL.USERINITFXN = prog.extern("EVMDM642_init");
%052 bios.RTDX.RTDXDATASEG = prog.get("SDRAM");
%053 // !GRAPHICAL_CONFIG_TOOL_SCRIPT_INSERT_POINT!
%054 if (config.hasReportedError == false) {
%055     prog.gen();
%056 }
```

上述的配置文件主要完成了以下任务：

处理器信息设置（003～010 行）；
DSP/BIOS 模块设置（012～016 行）；
内存配置（020～023 行）；
用户自定义设置（025～029 行）；
任务的声明与函数绑定（036～037 行）；
外设驱动设置（039～049 行）；
初始化设置（051 行）；
……

这里不详细介绍具体配置 tcf 文件的语法，可以参考《DSP/BIOS 5.30 Textual Configuration（Tconf） User's Guide》（SPRU007H）。实际上 TI 也在 Code Composer Studio 中已经提供了相当人性化的图形配置界面，如图 1.17 所示。通过 Gconf，绝大多数的配置选项都可以通过修改属性方式很方便地完成。

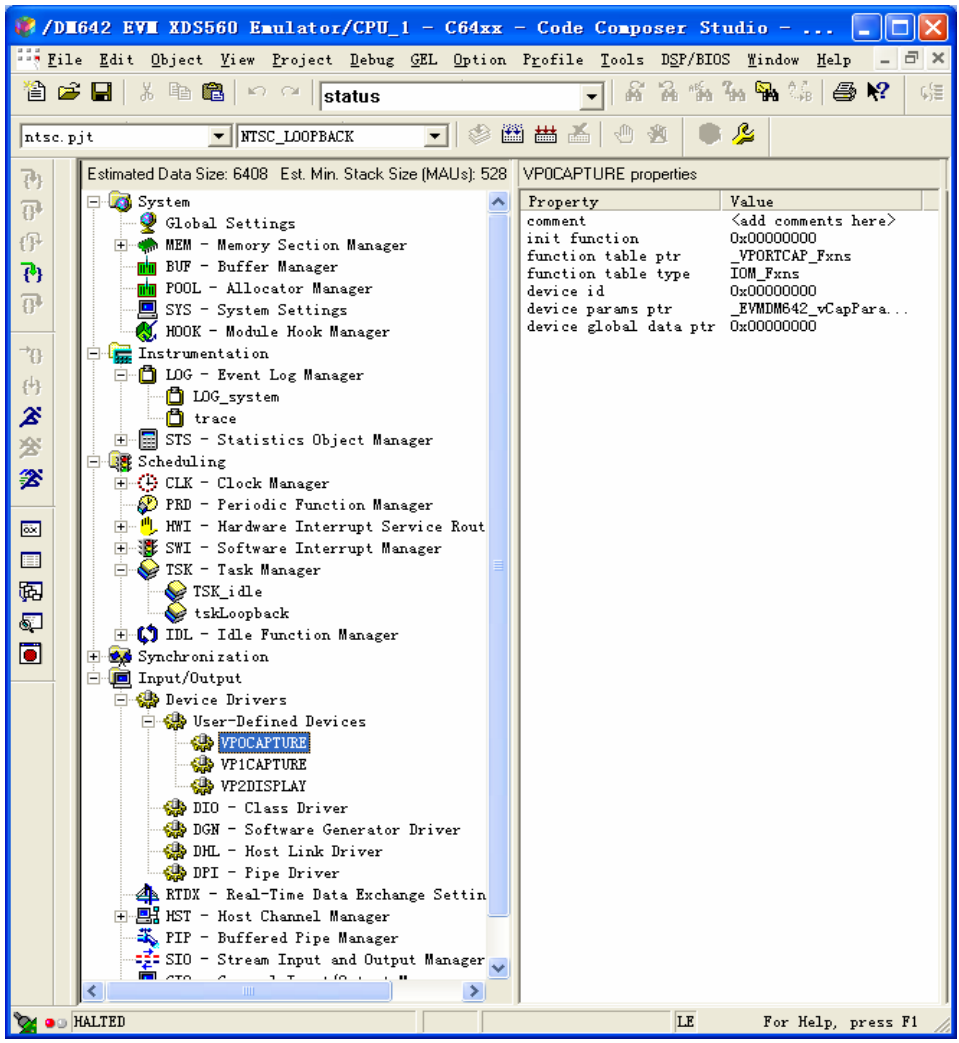


图 1.17　DSP/BIOS 图形配置界面

1.4.5　CCS（Code Composer Studio）

1. CCS 简介

CCS 是一个由 TI 公司提供的集成开发环境（IDE），它扩展了基本的代码产生工具，集成了 C 编译器、C 优化器、汇编器、汇编优化器和连接器等，并集成了调试和代码实时分析功能。开发者的一切开发过程都是在 CCS 这个集成环境下进行，包括项目的建立、源程序的编辑及程序的编译和调试。另外，CCS 还提供了更加丰富和强有力的调试手段来提高程序调试的效率和精度。图 1.18 为工作中的 CCS 示意图。

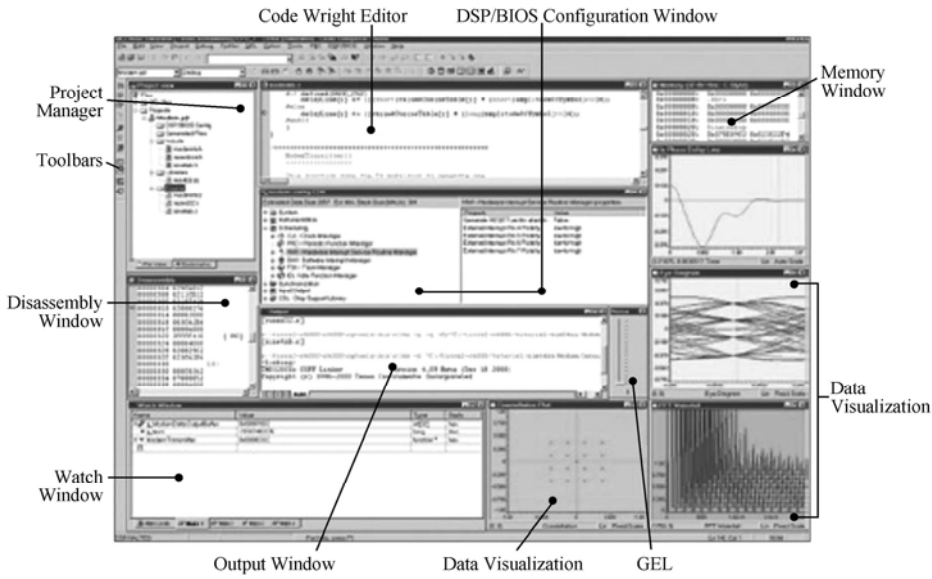


图 1.18　工作中的 CCS 示意图

2. 平台迁移问题

目前最新的 CCS 版本是 CCS3.3，DSP/BIOS 版本是 5.33.03。新版本的开发工具在功能上有很大增强，并且编译器也进行了更好的优化，在大多数情况下都推荐更新使用新版本。但 DM642 因为设计时间较早，配套的驱动文件基本上都是为 CCS2.X 或者 CCS3.0 所设计的，此外大量的现有代码是在旧版本的平台上开发完成，因此平台迁移是在 DSP 开发中经常要进行的一个工作。幸运的是 TI 公司的开发工具基本都保证了良好的向下兼容性，一般情况下的迁移工作都可以自动完成。当然还有一些细节仍需加以关注，如将 CCS3.0 下的项目文件迁移到新开发平台（CCS3.3 和 DSP/BIOS 5.33.03）时，应该注意：

（1）BSL（Board Support Library）需要重新编译。这主要是因为编译器升级导致的 CSL（Chip Support Library）接口发生变化，因此在移植到新的开发环境时需要将所有的 bsl 库针对现有平台重新编译。以 DM642 为例，本书使用的是 Spectrum Digital 提供的 Version 3 开发平台，原始所附的 evmDM642bsl.lib 为在 CCS3.0 下编译生成，并提供了源代码（\$CCSInstallDir\boards\evmdm642\ lib\evmDM642bsl\）。重新编译时需要将当前 CSL 的头文

件目录（如 C:\CCStudio_v3.3\C6000\csl\include）添加到项目 evmDM642bsl 的编译选项中去，然后以新生成的库文件替代原始的 bsl 库，这样才能保证对应的文件能够在 CCS3.3 下顺利连接。

（2）不同 DSP/BIOS 版本之间的切换。因为 DSP/BIOS 和底层紧密联系，因此，在使用新版本的 DSP/BIOS 时有可能出现向下兼容的问题，有时我们不得不使用低版本的内核。幸运的是 CCS3.3 提供了选择不同 DSP/BIOS 的机制，具体的选项在 Help→About→Component Manager 之中，如图 1.19 所示。

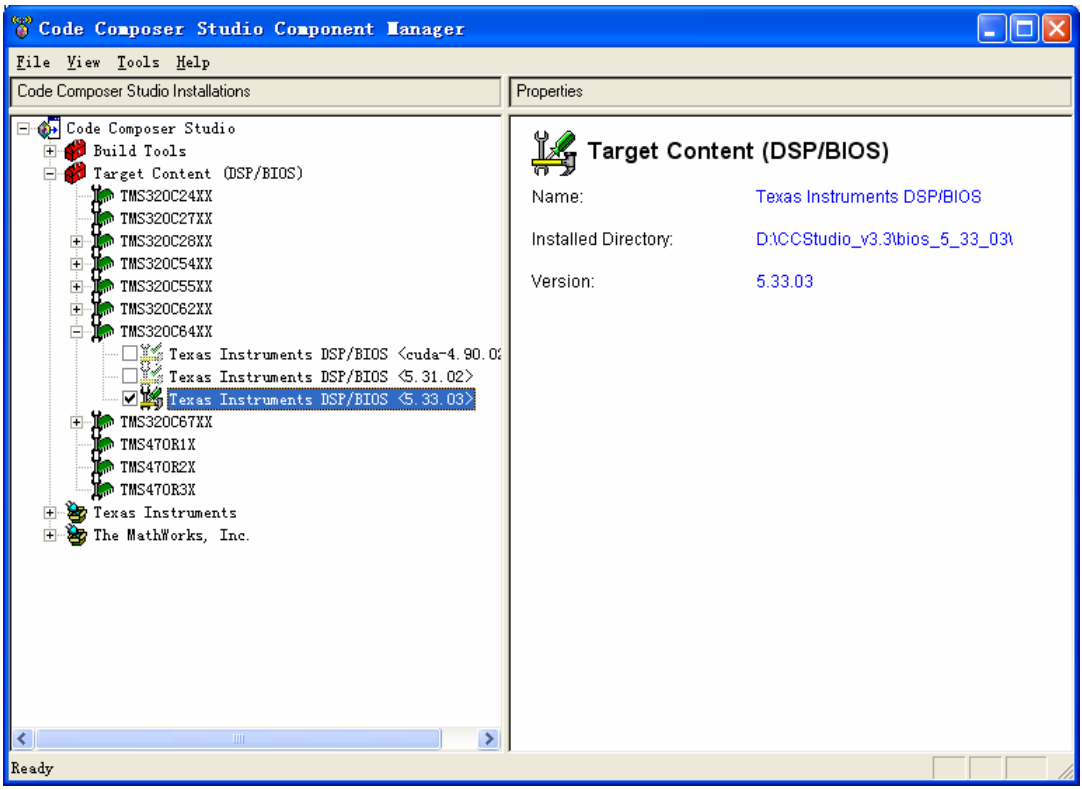


图 1.19　选择不同的 DSP/BIOS

1.4.6　软/硬件接口

下图为实际的 DM642 EVM 硬件板卡的实际硬件连接图，多媒体应用时，外围设备众多，如图 1.20 所示，注意正确地连接。

在连接硬件平台时需要特别注意连接顺序的问题[①]：

（1）在任意其他连线之前（包括视频口），首先插上 JTAG 仿真线。

（2）在连接显示器，摄像头的时候需要保持关闭状态。

（3）在其他设备都连接好的情况下再接通开发板的电源。

[①] 参见 http://c6000.spectrumdigital.com/evmdm642/v3/docs/evmdm642_faq.html

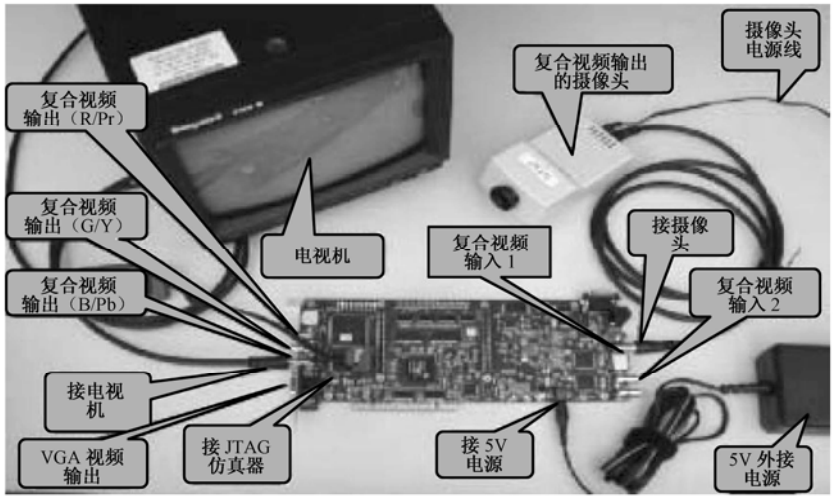


图 1.20　硬件平台连接

此外在硬件连接时应注意摄像头和显示器的制式（NTSC/PAL）及板上视频编解码器的种类，以便在程序中进行设置。

1.4.7　一个示例程序

下面通过一个简单的示例程序来演示如何通过使用 DSP/BIOS 和 Code Composer Studio 实现一个基本的图像处理任务，该程序项目如图 1.21 所示。

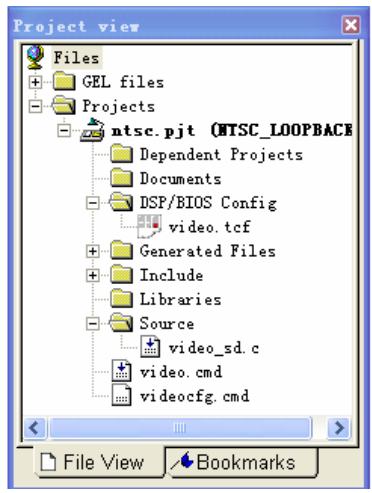


图 1.21　示例项目

在这个项目中我们希望直接从一个 NTSC 制式的摄像头中获取视频，然后在 NTSC 制式的电视机中显示出来。DSP/BIOS 的配置见图 1.17，其中建立了一个 tskLoopback 任务，这个任务由函数 tskVideoLoopback（video_sd.c）实现。

整个 video_sd.c 由三部分构成：静态变量声明、初始化 main 函数和 task 函数，下面分别讲解这三部分的内容。

1）静态变量声明

示例 1-5　全局声明：获得编解码器信息。

```
        %001 /* caputure configuration parameters */
        %002 /* embedded sync mode is recommended as it offers better re-sync
capability */
        %003 VPORT_PortParams EVMDM642_vCapParamsPort= EVMDM642_CAP_PARAMS_
PORT_EMBEDDED_DEFAULT;
        %004 static VPORTCAP_Params EVMDM642_vCapParamsChan= EVMDM642_CAP_
PARAMS_CHAN_EMBEDDED_DEFAULT(NTSC720);
        %005
        %006 /* For the vp0 & tvp5146, the port number must be fixed at 0 */
        %007 /* Analog input format can be either COMPOSITE or S-Video         */
        %008  static  TVP51XX_ConfParams  EVMDM642_vCapParamsTVP5146      =
EVMDM642_CAP_PARAMS_TVP51XX_EMBEDDED_DEFAULT(NTSC601, COMPOSITE, 0);
        %009
        %010 /* display configuration parameters */
        %011 VPORT_PortParams EVMDM642_vDisParamsPort= EVMDM642_DIS_PARAMS_
PORT_DEFAULT;
        %012 static VPORTDIS_Params EVMDM642_vDisParamsChan= EVMDM642_DIS_
PARAMS_CHAN_BT656_DEFAULT(NTSC);
        %013 static SAA7105_ConfParams EVMDM642_vDisParamsSAA7105= EVMDM642_
DIS_PARAMS_SAA7105_SDTV_DEFAULT(NTSC720, SVIDEO);
        %014
        %015 /* heap IDs defined in the BIOS configuration file */
        %016 extern Int EXTERNALHEAP;
```

注意这里编解码器用的是 SAA7105 与 tvp5146，输入、输出全部采用 NTSC 制式，这样就可以在静态声明部分通过 008 行和 012 行获得这两个元器件的相应参数，以便在后面的视频获取处理中使用。

2）main 函数

标准的 DSP/BIOS 将会执行以下初始化顺序：

（1）初始化 DSP。一个 DSP/BIOS 程序将会从指针 c_int00 开始（在 reset 时候置位），对 C6000 平台来说，c_int00 开始的代码将完成系统堆栈（由寄存器 B15 指向）和控制寄存器的初始化。

（2）从.cinit 块中对程序 bss 部分进行初始化。

（3）调用 BIOS_init 函数完成 DSP/BIOS 各模块的初始化。

（4）在所有模块初始化以后调用主程序的 main 函数。

（5）调用 BIOS_start 函数启动 DSP/BIOS。

下面是示例程序的 main 函数源代码。

示例 1-6　main 函数：初始化。

```
        %001 main()
```

```
%002 {
%003    /*******************************************************/
%004    /* open CSL DAT module for fast copy                   */
%005    /*******************************************************/
%006    CSL_init();
%007    CACHE_clean(CACHE_L2ALL, 0, 0);
%008    CACHE_setL2Mode(CACHE_256KCACHE);
%009    CACHE_enableCaching(CACHE_EMIFA_CE00);
%010    CACHE_enableCaching(CACHE_EMIFA_CE01);
%011    DAT_open(DAT_CHAANY, DAT_PRI_LOW, DAT_OPEN_2D);
%012 }
```

从代码中可以发现 main 函数主要是完成 DSP 与 DMA 初始化的工作，包括：

（1）清除 L2cache；

（2）设置缓存模式（008～010 行）；

（3）打开 DMA 通道（011 行）。

3）taskVideoLoopback 函数

随后系统进入 DSP/BIOS 控制阶段，task 按照优先级顺序调度。此时系统只有两个 task（见图 1.17）：TSK_idle 和 tskLoopbak, 其中 TSK_idle 为 IDL 任务，具有最低的优先级；处理器主要在运行 tskLoopbak 任务，它的实现函数为 tskVideoLoopback 函数，该函数也在 video_sd 中实现，代码如示例 1-7 所示。

示例 1-7　tskVideoLoopback：基本的帧处理。

```
%001 /*
%002  * ======== tskVideoLoopback ========
%003  * video loopback function.
%004  */
%005 void tskVideoLoopback()
%006 {
%007    Int status;
%008    FVID_Handle disChan;
%009    //Int frames = 0;
%010    FVID_Frame *disFrameBuf;
%011    Int numLinesDis = EVMDM642_vDisParamsChan.imgVSizeFld1;
%012    Int numLinesCap = EVMDM642_vCapParamsChan.fldYStop1 - EVMDM642_
vCapParamsChan.fldYStrt1+1;
%013     Int numLines = (numLinesDis > numLinesCap) ? numLinesCap :
numLinesDis;
%014    Int i;
%015    FVID_Handle capChan;
%016     Int numPixels=EVMDM642_vCapParamsChan.fldXStop1- EVMDM642_
vCapParamsChan.fldXStrt1+1;
%017    FVID_Frame *capFrameBuf;
```

```
        %018    Int capLinePitch = EVMDM642_vCapParamsChan.fldXStop1-EVMDM642_
vCapParamsChan.fldXStrt1+1;
        %019    Int disLinePitch = EVMDM642_vDisParamsChan.imgHSizeFld1;
        %020    numLines *= 2; /* both fields */
        %021
        %022    /*******************************************************/
        %023    /* allocate both capture and display frame buffers    */
        %024    /* in external heap memory                            */
        %025    /*******************************************************/
        %026    EVMDM642_vCapParamsChan.segId = EXTERNALHEAP;
        %027    EVMDM642_vDisParamsChan.segId = EXTERNALHEAP;
        %028    EVMDM642_vDisParamsSAA7105.hI2C = EVMDM642_I2C_hI2C;
        %029    EVMDM642_vCapParamsTVP5146.hI2C = EVMDM642_I2C_hI2C;
        %030
        %031    /*******************************************************/
        %032    /* initialization of capture driver                   */
        %033    /*******************************************************/
        %034     capChan = FVID_create("/VP0CAPTURE/A/0",IOM_INPUT, &status,
(Ptr)&EVMDM642_vCapParamsChan, NULL);
        %035
        %036    /*******************************************************/
        %037    /* initialization of display driver                   */
        %038    /*******************************************************/
        %039    disChan = FVID_create("/VP2DISPLAY", IOM_OUTPUT,&status, (Ptr)&
EVMDM642_vDisParamsChan, NULL);
        %040
        %041    /*******************************************************/
        %042    /* configure video encoder & decoder                  */
        %043    /*******************************************************/
        %044    FVID_control(disChan, VPORT_CMD_EDC_BASE + EDC_CONFIG,  (Ptr)&
EVMDM642_vDisParamsSAA7105);
        %045    FVID_control(capChan, VPORT_CMD_EDC_BASE + EDC_CONFIG, (Ptr)&
EVMDM642_vCapParamsTVP5146);
        %046
        %047    /*******************************************************/
        %048    /* start capture & display operation                  */
        %049    /*******************************************************/
        %050    FVID_control(disChan, VPORT_CMD_START, NULL);
        %051    FVID_control(capChan, VPORT_CMD_START, NULL);
        %052
        %053    /*******************************************************/
        %054    /* request a frame buffer from display & capture driver */
        %055    /*******************************************************/
```

```
        %056    FVID_alloc(disChan, &disFrameBuf);
        %057    FVID_alloc(capChan, &capFrameBuf);
        %058
        %059    while(1){
        %060     /* copy data from capture buffer to display buffer */
        %061     /*************************************************/
        %062     for(i = 0; i < numLines; i ++) {
        %063     DAT_copy(capFrameBuf->frame.iFrm.y1+i*capLinePitch,disFrameBuf
-> frame.iFrm.y1 + i * disLinePitch, numPixels);
        %064     }
        %065     DAT_wait(DAT_XFRID_WAITALL);
        %066     FVID_exchange(capChan, &capFrameBuf);
        %067     FVID_exchange(disChan, &disFrameBuf);
        %068    }
        %069  }
```

从上述代码中可以发现 001～058 行语句主要在完成初始化的工作，而实际的图像处理在 059～068 语句中完成，因为这里要完成的任务仅仅是将数据由接收缓冲区搬移到输出缓冲区，因此直接使用 DAT_copy（见 063 行）调用 DMA 来完成对每一帧的搬移。当然这仅是一个简单的范例，但从这个范例出发，我们可以实现一些基本的基于单帧的视频处理算法。

参 考 文 献

［1］陈玉，王宗和，张旭东．TMS320 系列 DSP 硬件开发系统．北京：清华大学出版社，2008．

［2］TI．TMS320C6000 Optimizing C/C++ Compiler User's Guide． SPRU1870．

［3］张旭东，卢国栋，冯健．图像编码基础与小波压缩技术．北京：清华大学出版社，2004．

［4］TI．TMS320C6000 DSP Enhanced Direct Memory Access Controller．Reference Guide．SPRU234．

［5］TI．DSP/BIOS 5.30 Textual Configuration(Tconf) User’s Guide，SPRU007H．

［6］魏振宇．H.264 快速算法研究及其在高速 DSP 平台上的实现［D］清华大学电子工程系，2005．

第 2 章　视频图像压缩编码基础

本章概述数字图像编码的一些主要技术，目前的视频编码标准主要以这些技术为核心。本章介绍的内容是理解各种视频编码标准的基础，这些标准包括 JPEG，H.261，H.263，H.264，MPEG-1，MPEG-2 和 MPEG-4。利用 DSP 实现视频压缩和处理，不管是使用专业公司提供的软件包还是自行开发软件，都应该对这些视频编码技术有一个基本的了解。对于理解目前的视频编码标准，这些介绍基本是够用的，对于希望全面深入学习视频编码技术的读者，可参考本章末给出的参考文献。

2.1　数字图像编码概述

数字视频的存储和传输需要大的存储容量或宽的传输信道，表 2.1 是几种常用视频图像格式的未压缩数据率，这无法在移动通信的环境中传输。即使在光网络环境下，当多用户同时使用时，也会造成网络的拥塞，在视频监控系统中，硬盘也无法长时间存储如此大量的数据，因此，对视频图像数据的压缩编码就成为一种迫切需求。正是这种需求，使得视频图像压缩编码算法和技术成为近年来非常活跃的一个研究和应用领域，并已取得极大的商业成功。

表 2.1　几种常见视频图像格式的未压缩数据率

视频源	每秒帧率	分辨率	未压缩数据率/（Mb/s）
NTSC	30	720×480	125
PAL	25	720×576	125
VCR	25	352×288	31
HDTV	30	1920×1080	1000

图像压缩的基本理论起源于 20 世纪 40 年代末期香农（Shannon）的信息理论。香农的编码定理告诉我们，在不产生任何失真的前提下，通过合理的编码，对于一个信源符号的表示，平均可以任意接近于信源的熵。在这个理论的框架下，出现了几种不同的无失真信源编码方法，如 Huffman 编码、算术编码、词典编码等，这些方法可以应用于一幅数字图像，获得一定的码率压缩，但无失真编码的压缩率是很有限的，对较复杂的自然图像，压缩率很难超过 2。

因为无失真信源编码的压缩率的限制，难以满足大多数图像存储和传输的需要。由于应用的需求，有失真压缩得到更广泛的研究。有失真压缩的目的是去除图像数据中的冗余信息和视觉不重要的细节分量，以尽可能少的码字来表示所处理的图像。

给定一幅数字图像，它的原始表示一般是空间像素阵列，这是它的空间域表示。在空间

域表示中，相邻的像素之间存在很强的相关性，冗余信息分布在较大范围的空间像素集中，直接处理比较困难。直观的思想是，通过一种变换，将图像从空间域映射到变换域中，在变换域可以进行简捷和有效的处理。对理想变换的第一种要求是将强相关的空间像素阵映射成完全不相关的、能量分布紧凑的变换系数阵，占少数的大的变换系数代表了图像中最主要的能量成分，占多数的小的变换系数表示了一些不重要的细节分量，通过量化去除小系数所代表的细节分量，用很少的码字来描述大系数所代表的主要能量成分，从而达到高的压缩比，这是用变换技术进行有失真编码能够达到高压缩比的主要原因；对于变换的第二种要求是，变换系数阵的物理含义要明确，容易与人们关于 HVS（Human Visual System，人类视觉系统）的知识相结合，以便有效去除视觉冗余，尽可能地保留对视觉重要的信息。

理想的去相关和保证能量紧凑的变换是 KL（Karhunen Loeve）变换，它使变换系数是统计不相关的，但 KL 变换的基不固定，由像素的相关系数矩阵的特征矢量列构成，由于特征分析的复杂性和需要存储变换基的额外开销，使得 KL 变换对于应用是不现实的。幸运的是，人们找到了 KL 变换的一个很好的逼近。对于强相关空间像素阵，人们发现 DCT（Discrete Cosine Transform，离散余弦变换）是 KL 变换的很好逼近，由于 DCT 具有固定的基和明确的物理含义，使得 DCT 广泛应用于图像压缩，成了变换编码方法的主要代表。

正是这个背景，20 世纪 80 年代中期开始制定的静止图像压缩编码的国际标准 JPEG 采用了 DCT 变换编码为其核心算法，并被广泛地接受和应用。在实际中，由于实现上和后处理的方便，图像被划分成 8×8 或 16×16 的小块，对每一个块进行单独的变换和后处理，这种块之间的单独处理带来了压缩效率上的限制和块效应问题，尤其当压缩倍数较高时，块效应（类似马赛克效应）成为 DCT 变换编码最主要的质量限制。

本章提供一些理解各种编码方法所需要的基础知识，在介绍了数字图像的基本表示方法之后，首先给出熵编码的基本概念和常用的实现算法，然后介绍量化的基本概念和优化算法，这些内容是各种编码方法共同需要的基础。讨论预测编码和变换编码是传统编码方法的主要内容，也是构成目前静止图像编码标准 JPEG 的基础算法。结合运动预测技术和 DCT 技术，介绍混合视频编码算法，这是构成当前视频序列编码国际标准的主要算法。

2.2　图像的表示和编码质量的评价

本节介绍两个最基本的预备知识，一是介绍在图像编码文献中常用的图像表示方法，或者说格式，主要介绍已数字化的图像表示格式，对于模拟图像数字化过程的分析不是关注的要点；二是介绍对已编码图像的几种常见质量评价方式。

2.2.1　静止图像格式

一幅静止的数字化图像是一个二维阵列，一共有 M 行，每一行有 N 个点，每一个点称为一个像素。一般情况下，对这个阵列定义一个参考点，通常将参考点定义在阵列左上角的像素位置上，设相邻像素之间的距离为 1，这样第 i 行，第 j 列上的像素值表示为 a（i，j）。那么，数据集合 $\{a(i,j), 0 \leqslant i < M, 0 \leqslant j < N\}$ 就表示一幅图像。

严格地说，如上的表示是一种简化了的方法，一幅真实的数字图像有多种不同类型，例

如，二值图像、单色图像（黑白图像）和彩色图像。对二值图像和黑白图像，a（i，j）完全表示一个像素，它表示像素的灰度值；但对彩色图像，每个像素都要由几个分量组成，单一的 a（i，j）表示不了一个彩色图像，但数据集合$\{a(i,j), 0 \leqslant i < M, 0 \leqslant j < N\}$可以表示其中的一个分量，一幅彩色图像可以看成是多个分量的组合，或者说是多个数据集合的组合。

彩色图像的每个像素由几个分量组成，最常用的分量组合方式有两种，RGB 和 YUV。其中 RGB 分别代表红、绿、蓝三基色，任何一种颜色均可由 R、G、B 三基色的不同取值混合而成。设红、绿、蓝的单位矢量用 $\boldsymbol{r}, \boldsymbol{g}, \boldsymbol{b}$ 表示，任何一种颜色可由 $\boldsymbol{r}, \boldsymbol{g}, \boldsymbol{b}$ 的组合构成，即：

$$\boldsymbol{C} = R \cdot \boldsymbol{r} + G \cdot \boldsymbol{g} + B \cdot \boldsymbol{b}$$

这里 $0 \leqslant R, G, B < 1$。

例如，$R=G=B=1$，构成白色；$R=G=B=0$，构成黑色；$R=G=1, B=0$，构成黄色等。

对于一幅彩色图像，数字化是两个方面的，第一方面，在空间均匀采样，获得 $M \times N$ 个样本阵列，即像素阵列；第二方面，对每个彩色分量数字化，即将 R, G, B 数字化。常用 8 位表示每一个彩色分量，模拟值 1 对应于数字化 255，模拟值 0 对应于数字化 0，中间值有均匀的对应关系，在数字图像表示中，仍用符号 R, G, B 表示数字化彩色分量，不会引起混乱。这样，一幅用三基色分量表示的数字图像，可以写为一个数据集合：

$$\left\{R_{i,j}, G_{i,j}, B_{i,j}, (0,0) \leqslant i, j < (M, N)\right\}$$

这种表示方法一般称为 RGB 表示，或称为 RGB 彩色坐标。

除了 RGB 分量表示以外，其他分量形式也用于彩色图像的表示，例如，YUV 格式，将图像的亮度和色度分开在不同分量中，由于各分量之间有更少的相关性，更适合于图像压缩的应用。在 YUV 格式中，Y 表示图像的亮度分量，U、V 代表它的色度分量，它们是色差分量。由 RGB 分量可以直接转换成 YUV 分量，它们的转换关系是：

$$Y = 0.299R + 0.587G + 0.114B$$

$$U = -0.147R - 0.289G + 0.436B = 0.492(B - Y)$$

$$V = 0.615R - 0.515G - 0.100B = 0.877(R - Y)$$

由于人的视觉系统对 UV 分量的空间分辨率的敏感性较低，因此，UV 分量经常在进行压缩编码之前就先进行亚采样，即在横向和纵向分别进行 2∶1 采样，只保留 1/4 的数据量，这有利于总体压缩比的提高。对彩色图像，人们常用的是对 YUV 各分量的压缩。

黑白图像和二值图像的表示只用一个分量，彩色图像压缩是针对各分量分别进行的，因此，在本书中，提到一个静止图像时，如果不加特殊说明，均指的是如下一个数据集合：

$$\{a(i,j), 0 \leqslant i < M, 0 \leqslant j < N\}$$

在实际应用中，静止图像的来源有几种，大多以计算机为存储和显示的载体。一种最主要的静止图像源来自扫描仪，它将照片和画册转换为计算机内的静止图像文件，目前的扫描仪一般会配备一个软件，将扫描的图像按 JPEG 标准压缩后存到计算机中；另一种静止图像的来源是数码照相机，它内嵌一个 JPEG 压缩编码器，将 CCD（Charge Coupled Device）输出的静止图像进行压缩后存储在机内的 Flash 存储器上，需要时，通过专用接口程序下载到计算机中存储和进一步处理；医学成像和其他传感设备也提供了有特殊应用要求的静止图像；电视信号中的某些固定帧，通过采集器输送到计算机中，也可以作为静止图像处理；传真机是典型的二值图像的例子。

2.2.2　视频序列的常用格式

由于视频序列是对电视信号或摄像机输出信号的数字化生成的，它是一种三维序列，在每一时刻，一幅完整的画面是一个二维的数字图像，由若干按时间顺序排列的二维数字图像构成三维的视频序列。可以用 $S(x, y, k)$ 表征一个视频序列，k 是表示时间顺序的序号。

原始的电视信号是一种扫描式的模拟信号，以 PAL 为例，它是按奇偶场扫描产生的模拟电视信号，每秒 50 场，一对奇偶场交叉形成一个图像帧，每秒 25 帧，每帧 625 行，其中可见行为 576，其他行在消隐区。因此，以帧为单位的电视信号在时间方向上本身就是离散信号，在每一帧中，在垂直方向也是离散的，对电视信号的离散化主要是对每一行进行采样。一个采样后的数字视频序列表示为 $S(x, y, k)$。

目前国际上存在几种不同的电视制式，主要分成 625 线和 525 线两种，ITU 制定了一个统一的标准来规范不同标准的电视信号的数字化，这就是 ITU-R BT.601。该标准采用 YC_bC_r 彩色坐标，YC_bC_r 是 YUV 格式的一个经过校正的版本，它与 RGB 坐标的基本变换关系为：

$$Y = 0.299R + 0.587G + 0.114B$$

$$C_b = -0.169R - 0.331G + 0.5B$$

$$C_r = 0.5R - 0.419G - 0.081B$$

数字化后用 8 位无符号二进制数表示，其中 Y 的取值范围为（16，235）。色度分量的取值范围为（16，240），128 代表零值。按照这个范围定义的数字化 YC_bC_r 分量表示为：

$$\overline{Y} = 219Y + 16$$

$$\overline{C}_b = 126(B - Y) + 128$$

$$\overline{C}_r = 160(R - Y) + 128$$

ITU-R BT.601 规定按统一的 13.5 MHz 对亮度信号进行采样，用 6.75 MHz 对色度信号进行采样，按帧表示图像的分辨率，表 2.2 给出 ITU-R BT.601 的主要参数指标。

表 2.2　ITU-R BT.601 的主要参数指标

编 码 信 号	525 行系统， 30f/s	625 行系统，25 f/s
每行采样数目 亮度信号 色度信号	 858 429	 864 432
每行有效像素数 亮度信号 色度信号	 720 360	 720 360
每帧有效行数	486	576
采样频率 亮度信号 色度信号	 13.5 MHz 6.75 MHz	 13.5 MHz 6.75 MHz
编码方式	8 位 PCM 编码	8 位 PCM 编码
量化值范围 亮度信号 色度信号	 16～235（220 个量化层） 16～240（224 个量化层）	 16～235（220 个量化层） 16～240（224 个量化层）
采样格式	4∶2∶2	4∶2∶2

表 2.2 的其他项目是清楚的，例如，对 PAL 制电视信号，按 ITU-R BT.601 格式数字化成

720×576 分辨率、每秒 25 帧的数字视频序列。这里对采样格式进一步说明。由于色度分量比亮度分量采样率低，怎样选择色度样本点与亮度样本点的位置关系，就是表 2.2 的采样格式问题。ITU-R BT.601 中，色度分量是亮度分量采样率的一半，因此采用在每行中隔一个亮度样点保留一个色度样点的方式，这就是 4∶2∶2 采样格式。其他常用的采样格式是：4∶4∶4、4∶1∶1 和 4∶2∶0，用于不同的标准中。这 4 种采样格式的示意图见图 2.1，在图中，小的实心圆表示亮度采样点，大的空心圆代表色度采样点，C_b 和 C_r 的采样点是重合的。

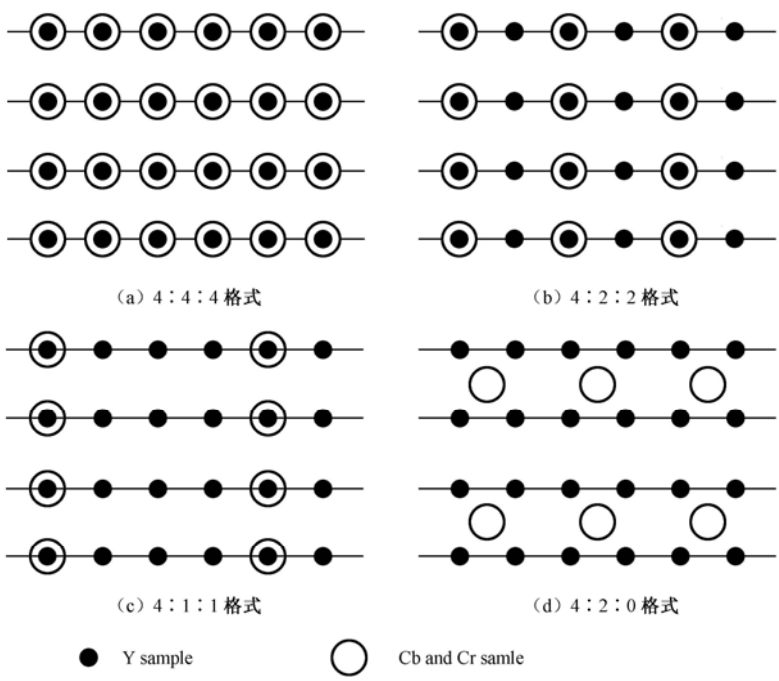


图 2.1　几种典型的视频图像采样格式

在实际中还常用更低分辨率的视频序列，它们可以由对 ITU-R BT.601 分辨率的亚采样获得。最常用的两种格式是 CIF 和 QCIF，它们的主要参数如表 2.3 所示。

表 2.3　CIF 和 QCIF 格式的主要参数

参　　数	CIF	QCIF
分辨率	352×288	176×144
亮度分量	176×144	88×72
色度分量		
采样格式	4∶2∶0	4∶2∶0

目前，在各种网络资源中，可以找到大批专用于视频编码研究的标准测试序列，这些序列大多是按 ITU-R BT.601、CIF 或 QCIF 格式存储的。目前制定的各种视频编码标准也是支持多种不同的视频格式，而实际的视频处理系统则是通过数字解码器把复合视频数字化成数字视频图像信号，或通过数字摄像机直接获得数字视频信号。

2.2.3　编码质量的评价

对于有失真的压缩算法，应该有一个评价准则对压缩后的图像质量给予评判，常用的评价准则有两种：一是客观准则，二是主观准则。

客观准则是对压缩还原后的图像与原始图像的误差进行定量计算，一般是对整个图像或图像中一个指定的区域进行某一种平均计算，以得到均值误差。

设一个原始图像为 $\{a(i,j),0\leqslant i\leqslant M-1,0\leqslant j\leqslant N-1\}$，相应压缩后的还原图像为 $\{\hat{a}(i,j),0\leqslant i\leqslant M-1,0\leqslant j\leqslant N-1\}$，误差图像为：

$$\{e(i,j)=a(i,j)-\hat{a}(i,j),\quad 0\leqslant i\leqslant M-1,0\leqslant j\leqslant N-1\}$$

那么，均方误差表示为：

$$e_{\text{mse}}=\frac{1}{MN}\sum_{i=0}^{M-1}\sum_{j=0}^{N-1}e^2(i,j)$$

有时也会用均方根误差表示，它是 $e_{\text{rms}}=\left[e_{\text{mse}}\right]^{1/2}$。

更常用的是信噪比表示，它用分贝表示压缩图像的定量性能评价，基本信噪比定义为：

$$\text{SNR}=10\lg\left[\frac{\sum_{i=0}^{M-1}\sum_{j=0}^{N-1}a^2(i,j)}{\sum_{i=0}^{M-1}\sum_{j=0}^{N-1}\left(a(i,j)-\hat{a}(i,j)\right)^2}\right]$$

另一种信噪比的定义是对原始图像首先去均值，定义如下：

$$\bar{a}(i,j)=\frac{1}{MN}\sum_{i=0}^{M-1}\sum_{j=0}^{N-1}a(i,j)$$

$$\text{SNR}_m=10\lg\left[\frac{\sum_{i=0}^{M-1}\sum_{j=0}^{N-1}\left[a(i,j)-\bar{a}(i,j)\right]^2}{\sum_{i=0}^{M-1}\sum_{j=0}^{N-1}\left(a(i,j)-\hat{a}(i,j)\right)^2}\right]$$

文献中最常用的是峰值信噪比（PSNR），设 $a_{\max}=2^K-1$，K 是表示一个像素点用的二进制位数。

$$\text{PSNR}=10\lg\left[\frac{MNa_{\max}^2}{\sum_{i=0}^{M-1}\sum_{j=0}^{N-1}\left(a(i,j)-\hat{a}(i,j)\right)^2}\right]$$

在许多采集的视频序列和商用图像的应用中，最常用的是取 $K=8$，所以在一些文献中，直接将 $a_{\max}=255$ 代入上式。

对压缩图像质量的第二种评价方式是主观评价，它是选择一组评价者对评价图像进行打分，对这些主观打分求平均，获得一个主观评价分，表 2.4 是两种典型的评分标准。

表 2.4　对图像质量的主观评分标准

得分	第一种评价标准	第二种评价标准
5	非常好	感觉不到失真
4	好	感觉到失真，但没有不舒服的感觉
3	一般	稍感觉到不舒服
2	较差	不舒服
1	差	感觉非常不舒服

每一种得分记为 C_i，每一种得分的评分人数为 n_i，一个被称为平均感觉分 MOS（Mean Opinion Score）的主观评价得分为：

$$\mathrm{MOS}=\frac{\sum_{i=1}^{k} n_i \cdot C_i}{\sum_{i=1}^{k} n_i}$$

例如，一段视频节目的评分为 4.5，这说明图像质量是相当好的。

评价图像压缩效果的另外一个重要指标是压缩比 C，它指的是表示原始图像每像素的比特数与压缩后平均每像素比特数的比值，也常用每像素比特值（bpp）来代表压缩效果。在视频图像序列传输的应用中，由于信道的限制，目标传输速率是确定的，例如，384 kb/s 表示每秒 384 kb，这时的压缩比就要根据所传输视频的帧率和分辨率来确定。

2.3　信息理论基础和熵编码

信息理论是图像编码的主要基础之一，它给出了对符号序列进行无失真表示所需要的比特率下界，为逼近这个下界而提出一系列熵编码算法。本节直观性地给出信息理论及熵编码的概要介绍，并无严格证明，对信息理论的详细讨论参考文献[2]。

2.3.1　离散信源的熵表示

在数字图像和其他一些信源中，表示信源的符号集是有限集，例如，由 8 比特表示的数字图像，它的符号集是从 0～255，共有 256 种符号，一般的英文文字由 128 种符号构成，一幅传真的黑白图像可抽象成由 0 和 1 两种符号构成。这种由有限符号集构成的信源，称为离散信源。一个具体的离散信源是一个随机变量序列或阵列，在每一时刻或位置，一个随机变量的取值来自信源的符号集。

假设一个离散信源的符号集由 N 个符号$\{x_1, x_2, \cdots, x_N\}$构成，在信源中，每个符号的出现概率是确定的，即存在一个概率分布表$\{p_1, p_2, \cdots, p_N\}$，并且满足$\sum_{k=1}^{N} p_k = 1$。这个概率表的含义是：信源在任一时刻或位置的输出 X 取 x_k 的概率为 p_k，记为：

$$p_k = P(X = x_k) = p(x_k)$$

对于一个离散信源，常分为两种类型考虑，一是无记忆信源，即信源的当前输出与以前

的输出是统计独立的，否则就称为有记忆的。

下面讨论离散信源的定量描述。

假设在某时刻，信源输出 $X=x_k$，即 x_k 这个符号出现，这个符号出现给接收者带来多少信息呢？一个很直观的理解是：一个概率小的符号出现将带来更大的信息量，也就是说信息量与该符号概率倒数成正比的，直观地定义符号 x_k 包含的信息或称为自信息量为：

$$I(x_k)=\mathrm{lb}1/p_k=-\mathrm{lb}p_k$$ ①

由 N 个符号组成的符号集$\{x_1, x_2,\cdots, x_N\}$构成的离散信源的每个符号的平均自信息量为：

$$H(X)=-\sum_{k=1}^{N}p_k\cdot\mathrm{lb}p_k \tag{2.1}$$

这个平均自信息 $H(X)$也称为信源的熵（Entropy）。

可以看到，取以 2 为底的对数是合理的，这通过下面的例子来说明。

示例 2-1　设信源由 a，b 两个符号构成，$p(a)=p(b)=1/2$，因此，各符号的自信息为：$I(a)=I(b)=-\mathrm{lb}1/2=1$

熵为 $H(X)=-\sum_{k=1}^{2}p_k\cdot\mathrm{lb}p_k=1$

符号 a 和 b 分别用码字 0 和 1 表示，每个符号用 1 比特表示。以 2 为底的对数定义的自信息的单位就称为比特。

设另一信源由 a，b，c，d 共 4 个符号组成，且 $p(a)=p(b)=p(c)=p(d)=1/4$，则有：

$I(a)=I(b)=I(c)=I(d)=2$ b，$H(X)=2$

如果分别用码字 00，01，10，11 表示（编码）a，b，c，d 这 4 个符号，每个符号占 2 比特，平均码长也是 2 比特。

设一信源由 a，b，c，d 这 4 个符号构成，各符号概率分布如表 2.5 所示。

表 2.5　符号概率分布 1

符号	a	B	c	d
P	1/2	1/4	1/8	1/8

则有 $I(a)=1$，$I(b)=2$, $I(c)=I(d)=3$

$$H(X)=\frac{1}{2}\times1+\frac{1}{4}\times2+\frac{1}{8}\times3+\frac{1}{8}\times3=1.75$$

给出 2 种编码方法，第一种每个符号的平均码长为 2；第二种编码方法用 0 表示符号 a，10 表示符号 b，110 表示符号 c，111 表示符号 d，这样，平均每个符号码字长为：

$$l=\sum_{k=1}^{N}p_k\cdot l(x_k)=\frac{1}{2}\times1+\frac{1}{4}\times2+\frac{1}{8}\times3+\frac{1}{8}\times3=1.75$$

这时平均码长等于信源的熵。

若 4 个符号的概率分布如表 2.6 所示。

① $\mathrm{lb}P_k=\log_2P_k$

表 2.6　符号概率分布 2

符号	*A*	*B*	*c*	*d*
P	0.4	0.3	0.18	0.12

则有 $I(a)$=1.321 9，$I(b)$=1.737 0，$I(c)$=2.473 9，$I(d)$=3.058 9，$H(X)$ =1.862 2，用上面给出的第 2 种码字分配，平均码长为：

l=0.4×1＋0.3×2＋0.18×3＋0.12×3＝1.9，这里 $l > H(X)$ 。

观察以上例子，可以得到几点有益的启示：

（1）平均码长 $l \geqslant H(X)$ 。

（2）如果所有 $I(x_k)$为整数，则令 $l(x_k)=I(x_k)$，可以达到使平均码长等于熵，一般情况下，通过较好的码字分配，可以使平均码长接近熵。

（3）对于非等概率分布符号集，一般更好的码字分配是不等长编码。

（4）等概率分布有更大的熵，即更大的不确定性，需要更长的平均码字。实际上，可以证明：

$$H(X)=-\sum_{k=1}^{N} p_k \cdot \mathrm{lb} p_k \leqslant \mathrm{lb} N \left(N=-\sum_{k=1}^{N} \frac{1}{N} \mathrm{lb} \frac{1}{N}\right) \tag{2.2}$$

这些观察并不全面，但可以帮助我们理解香农的编码定理。基于对信源熵的表示，香农给出了离散信源的编码定理，这个编码定理给出了对离散信源编码的界，也即最有效编码方法所能达到的目标。下面我们先简要介绍条件熵和块熵，然后讨论信源编码定理。

设 X、Y 是两个随机变量，它们可能是信源的两个连续输出，也可能是两个有关系的不同信源的输出，在已知 Y 的观察值情况下，X 的条件熵为：

$$\begin{aligned} H(X \mid Y) &= \sum_{i=1}^{n} \sum_{j=1}^{m} P\left(x_i, y_j\right) \log \frac{1}{p\left(x_i \mid y_j\right)} \\ &= -\sum_{i=1}^{n} \sum_{j=1}^{m} P\left(x_i, y_j\right) \log p\left(x_i \mid y_j\right) \end{aligned} \tag{2.3}$$

可以证明 $H(X) \geqslant H(X \mid Y)$，也就是说，在已知 Y 的观察值情况下，X 的熵不会增加。

设 X_1，X_2，…，X_n 构成一个随机变量块，例如，它们可能是数字图像中一个块内的像素值，如果已知联合概率分布 $p\left(x_1, x_2, \cdots, x_n\right)=P\left(X_1=x_1, \cdots, X_n=x_n\right)$，块熵定义为：

$$H(X_1, X_2, \cdots, X_n)=-\sum_{x_1} \sum_{x_2} \cdots \sum_{x_n} p\left(x_1, x_2, \cdots, x_n\right) \log p\left(x_1, x_2, \cdots, x_n\right)$$

由 $p\left(x_1, x_2, \cdots, x_n\right)=p(x_1) p(x_2 \mid x_1) p(x_3 \mid x_1 x_2) \cdots p(x_n \mid x_1 x_2 \cdots x_{n-1})$

可以得到：

$$\begin{aligned} &H(X_1, X_2, \cdots, X_n)=H(X_1)+H(X_2 \mid X_1)+\cdots+H(X_n \mid X_1 X_2 \cdots X_{n-1}) \\ &=\sum_{k=1}^{n} H(X_k \mid X_1 X_2 \cdots X_{k-1}) \leqslant \sum_{k=1}^{n} H(X_k) \end{aligned} \tag{2.4}$$

上式最后一行利用了 $H(X) \geqslant H(X \mid Y)$ 。当各变量统计独立时，上式等号成立。这些关系式的应用和解释，我们将在讨论了编码定理后再进一步分析。

2.3.2　信源编码定理

所谓信源编码就是用二进制码字序列表示一个信源的输出序列。为了简单，先考虑离散无记忆信源，假设信源输出为 X，它的熵为 $H(X)$，编码后，平均每个信源输出符号用 R 比特表示，定义 ρ 为编码效率，这里有：

$$\rho = H(X)/R \tag{2.5}$$

这个编码效率定义的合理性很快就会看到。

首先考虑使用定长码字，即信源的每一个符号都采用相同长码字，对于一个有 N 个符号的信源，如果 $N=2^L$，则码长为 $R_F=\text{lb}N=L$。如果 N 不满足 2 的整数幂的条件，要唯一表示 N 个符号，码长 R 必须满足：

$$R_F = \text{lb}N + 1 \tag{2.6}$$

下面我们仅讨论 N 不为 2 的整数次幂的一般情况。

由于 $H(X)\leqslant \text{lb}N$，因此 $R_F \geqslant H(X)$，定长编码不可能达到比 $H(X)$更低的码率。

对这个问题做一点扩充，不是一次单独编码一个输出符号，而是编码一个块，即将信源的 k 个连续输出构成一个符号块进行编码，一个块作为一个独立编码单位，它的符号集的数目为 N^k，因此块的码字长 R_b 满足：

$$R_b = \text{lb}N^k + 1 = k\text{lb}N + 1$$

平均一个输出符号的码长为：

$$R_F = R_b/k = \frac{1}{k}\lfloor k\,\text{lb}N \rfloor + \frac{1}{k} \geqslant H(X)$$

尽管 $R_F \geqslant H(X)$仍然保持，但是块编码的效率比单符号编码有提高。

示例 2-2　一个符号集有 10 个符号，其中有 2 个符号出现的概率为 1/5，4 个为 1/10，4 个为 1/20。可以计算出其熵为 $H(X)=3.121\,9$。如果每个符号单独用定长码，则 $R_F=4$，编码效率为 $\rho=78.05\%$。如果每 8 个符号组成一个块进行编码，则码长为：

$R_F = [81\text{b}10]/8 + 1/8 = 3.375$，编码效率提高为 $\rho=92.5\%$。

由此可见，块编码可以提高编码效率，但仍以 $H(X)$为限，这实际就是定长码编码定理的内容。

定长码编码定理：设 X 是离散无记忆信源的输出，$H(X)$是其熵，对 X 能够准确解码的定长编码的码长 $R\geqslant H(X)$。

直观地考虑，对于一些经常出现的符号，采用较短的码字分配，对于一些较小概率出现的符号，分配较长的码字，可以使平均码长降低，例 1 也说明了这点，因此，实际上更常用的编码方法是变长编码，也就是熵编码。

与定长编码比，变长编码更复杂，除唯一可解码的要求，还存在即时解码问题，下面通过例子进行分析。

在例 1 中，给出一个由 a，b，c，d 4 个符号组成的信源，并给出一种变长码分配，这个码字分配称为编码 I，表 2.7 中同时也列出另外一种码字分配，称为编码 II 。

表 2.7　一个信源的两种编码

符 号	概 率	编 码 I	编 码 II
a	1/2	0	0
b	1/4	10	01
c	1/8	110	011
d	1/8	111	111

编码 I 是可即时解码的，当收到一个完整码字时，它可以立即恢复其所代表的符号。编码Ⅱ是可以唯一解码的，但却不是即时的，例如，得到码串 01 时，不能立即解码为 *b*，要待下一个比特到来，如果下一个比特是 0，前面的码字 01 被解码为 *b*，如果下一个比特为 1，011 一起解码为 *c*。编码 I 的特点是，对于一个码字 C_k，其码长为 *k*，不存在另外一个长度为 $1 \leqslant l \leqslant k-1$ 的码字，其二进制表示正好与 C_k 的前 *l* 比特是一致的，符合这个条件的编码，称为符合“前缀条件”，是可以即时解码的，容易看出，编码Ⅱ不符合这个条件。可即时解码有着应用上的简单性。

如果用树形式表示码字分配，如图 2.2 所示，可以看到，编码 I 的每一个码字都是树的终端节点，但编码Ⅱ不是。

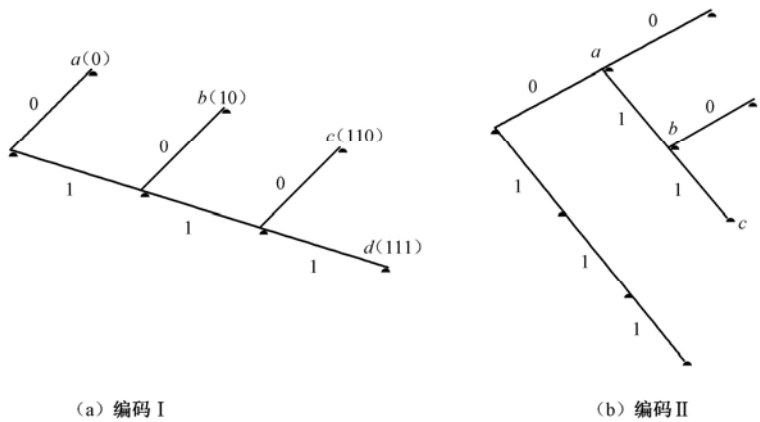


图 2.2　编码树

知道了前缀码的好处，变长的前缀码可以达到什么样的编码效率呢？如下的定理给出了回答。

变长码编码定理：设 *X* 是离散无记忆信源的输出，信源熵为 *H*(*X*)，信源有有限的输出符号，可以构造一个符合前缀条件的唯一可解的编码，使得平均码长 R_V 满足 $H(X) \leqslant R_V < H(X)+1$。

该定理给出了变长编码效率的界，平均码长总是大于等于信源的熵，但总可以找到一个编码，使码长小于信源熵加 1。同样，块编码可以增加变长编码的有效性，尤其信源输出符号集较小的情况，对于无记忆信源，*k* 个符号组成块进行编码，*k* 个符号组成的块的熵是 $k \cdot H(X)$，因此，块编码的码长为 $k \cdot H(X) \leqslant R_{bv} < k \cdot H(X)+1$，每个符号的平均码长为：

$$H(X) \leqslant R_V < H(X) + \frac{1}{k}$$

用编码定理，只能得到编码效率的一个下限，实际中，合理选择的变长编码总会得到比固定长编码更高的效率，后面几节给出变长编码的一些具体实现算法。

2.3.3　Huffman 编码

Huffman 编码是一种形成前缀变长编码的方法，它根据信源中每个符号 x_k 的出现概率 p_k 进行码字分配，出现概率最小的分配最长码字。先将信源中各符号出现概率按大小排成一列，出现概率最大的符号用 x_1 表示，放在最顶部，出现概率最小的符号用 x_N 表示，放在最底部，这个原始符号列构成最左边的一列，然后，从左向右进行码字分配，具体程序如下。

首先从最底端取 x_N 和 $x_{N\text{-}1}$ 构成一个偶，$x_{N\text{-}1}$ 作为上分支，x_N 作为下分支，每个分支分别赋予 0 和 1 码，然后以 $p(x_N)+p(x_{N\text{-}1})$ 所得概率构成一个新的符号 x'_{N-1}，这样 $\{x_1,x_2,\cdots,x_{N-2},x'_{N-1}\}$ 组成新符号组，从新符号组中选最低概率的两个符号组成一个偶，这两个最低概率符号可能是 x_{N-2},x'_{N-1}，也可能是 x_{N-3},x_{N-2}，为了叙述简单，我们假设是 x_{N-2},x'_{N-1}，它们向右引出上、下两个分支，每个分支分别赋于 0 和 1 码，并且以 $p(x_{N-2})+p(x'_{N-1})$ 为概率构成一个新符号 x''_{N-2}，组成新的符号集 $\{x_1,x_2,\cdots,x_{N-3},x''_{N-2}\}$，将这个过程递推下去，直到剩下最后 2 个符号，构成一对偶，形成上、下 2 个分支，分别赋于 2 个分支 0 和 1 码，结果实际上构成了一棵树，从最右边起，通过各分支到达最左边的符号集，每个符号对应于树的一个终端节点，将分支码从右向左串起来，构成一个码字，这个码字就赋于这个终点的符号，因为每个符号都是一个终端节点，因此，构成的编码是符合前缀条件的，这个过程中，由于概率最小的符号节点，经历从右向左的分支数最多，因此赋予了最长的码字。

下面通过例子进一步说明。

示例 2-3　图 2.3 所示的表给出一个有 8 个符号的信源，按概率从大到小排列，各符号的出现概率和自信息也列在表中，最后两列是编码后的码长和码字，Huffman 编码树在表下面，过程很清楚。

图 2.3 中，x_7，x_8 构成一个偶，它们向右引出两个分支，分别分配码 0 和 1。它们的概率和为 0.02，以 0.02 为概率构成新符号 x'_7，x'_7 和 x_6 是概率最低的 2 个符号，它们构成新的偶，向右引出两个分支，并分别分配码 0 和 1，这个过程一直进行到只有两个符号 x_1 和 $x_2^{(6)}$，分别为概率 0.4 和 0.6，它们构成最后两个分支，分别赋于码 0 和 1，最终构成一棵树，从树根到 x_1 只有一个分支，码串为 0，到 x_2 有两个分支，码串为 10，到 x_8 有 7 个分支，码串为 111 111 1。通过计算可以得到信源熵 $H(X)$＝2.272 2，平均码长 R_V=2.32，编码效率为 ρ＝97.94%。

符号	P_K	自信息	码长	码字
x_1	0.4	1.321 9	1	0
x_2	0.25	2.0	2	10
x_3	0.15	2.737	3	110
x_4	0.1	3.321 9	4	111 0
x_5	0.05	4.321 9	5	111 10
x_6	0.03	5.058 9	6	111 110
x_7	0.01	6.643 9	7	111 111 0
x_8	0.01	6.643 9	7	111 111 1

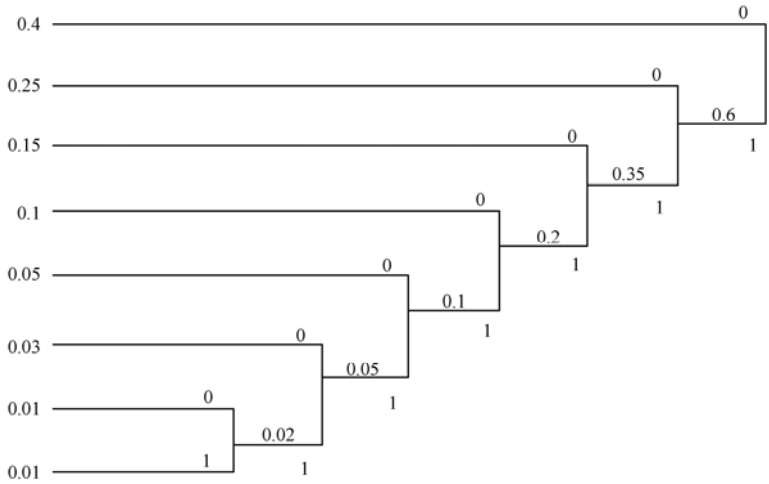


图 2.3　　Huffman 编码实例 1

示例 2-4　示例 2-3 是个很特殊的例子，任何一个偶组成的新符号，与原符号集排在最近的一个符号一起构成新的偶，编码次序很清楚，示例 2-4 稍复杂一点，但记住最关键的一点，每一步总是最低概率的两个符号构成一个偶。

在第一步，由 x_6,x_7 构成一个偶，新符号 x_6' 的概率为 0.1，当前符号集是 $\{x_1,x_2,\cdots x_5,x_6'\}$，其中 x_5 和 x_6' 为最低概率，由它们构成一个新的偶，得到新符号 x_5''，其概率为 0.18，当前符号集变为 $\{x_1,x_2,\cdots x_4,x_5''\}$，在这个符号集中，概率最小的是 x_3,x_4，分别为 0.12 和 0.15；下一步，x_3,x_4 构成一个偶，得到新符号 x_3''，构成新符号集 $\{x_1,x_2,x_3''',x_5''\}$，概率分布为 {0.35,0.2,0.27,0.18}；下一步，x_2 和 x_5'' 构成一个偶，并得到新符号 $x_2^{(4)}$，它的概率为 0.38，新符号集为 $\{x_1,x_2^{(4)},x_3'''\}$，概率分布为{0.35,0.38,0.27}；下一步，x_1 和 x_3'' 构成一个偶，得到新符号 $x_1^{(5)}$，其概率为 0.62；最后的一步 $x_1^{(5)}$ 和 $x_2^{(4)}$ 构成一个偶。每次形成一个偶，分别形成上、下两个分支，并赋予码 0 和 1，图 2.4 表示了这一过程，这个信源的熵 H（X）=2.492 9，平均码长 R_V=2.55，编码效率 ρ＝97.76%。

符　号	P_K	自 信 息	码　长	码　字
x_1	0.35	1.514 6	2	00
x_2	0.20	2.321 9	2	10
x_3	0.15	2.737	2	010
x_4	0.12	3.058 9	3	011
x_5	0.08	3.643 9	3	110
x_6	0.06	4.058 9	4	111 0
x_7	0.04	4.643 9	4	111 1

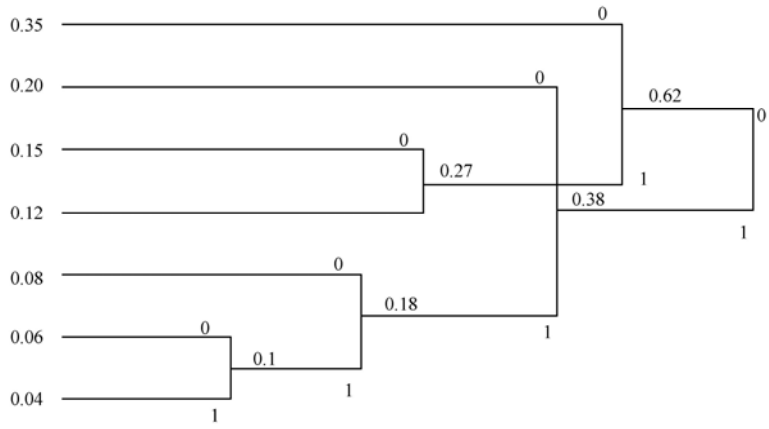


图 2.4　Huffman 编码实例 2

从以上两个例子可以看到，Huffman 编码是非常简单和有效的，并且是一种可唯一和即时解码的熵编码方法。有两点需要注意，一是 Huffman 编码的码字分配不是唯一的，每一步的两个分支中，哪一个分支配 0，是任意的，可以组成不同的码字，例如，在示例 2-3 中，将码字中的 1 和 0 互换，仍是唯一和即时解码的；另一点需要注意的是，尽管 Huffman 编码是很有效的，但无记忆信源的块 Huffman 编码会更有效。

以上过程是根据信源模型–概率分布构造 Huffman 码表的过程，在 Huffman 码表构成后，信源的编码和解码过程就是一件简单的事情了。对于一个信源输出序列，每得到一个符号，从 Huffman 码表中找到相应的码字，并输出到码流中，这个过程连续执行，得到一个输出序列的 Huffman 编码。解码过程也很简单，由于 Huffman 码符合前缀条件，因此，它是唯一和可即时解码的，解码器保持一个缓冲器，用于存放从码流中收到的比特，一开始缓冲器是空的，每收到一个比特，将它按次序压入缓冲器，并将缓冲器中已形成的字与 Huffman 码表中的每一个码字比较，如果找到一个相同的，输出该码字对应的信源符号，并将缓冲器刷新成空，否则，继续读取码流中的下一个比特，这个过程一直继续到结束。

例如，示例 2-4 的信源输出一个符号序列 $x_3x_1x_2x_1x_8x_2x_1$，通过查找 Huffman 码表，这个序列的编码是 110，0，10，0，111 111 1，10，0。解码器起始状态时，缓冲器空，收到第一个比特并压入缓冲器的为 1，码表中没有这个码字，继续读入第 2 比特，缓冲器变为 11，码表中仍没有该码字，继续读入第 3 比特，缓冲器变为 110，查找到与码表中符号 x_3 的码字相同，因此，得到第一个符号 x_3，缓冲器刷空，解码过程继续，直到完全恢复原始序列。

2.3.4　算术编码

Huffman 编码保持整数比特的码字分配，在这种约束下，它是一种最优编码，如果信源中各符号出现的概率是 1/2 的各次幂，则 Huffman 编码可以达到信源的熵，但这在实际中一般是不可能的。Huffman 编码的另一种限制对二元信源是无效的，即如果一个信源由 2 个符号组成，Huffman 编码只能分配 0 和 1 两个码字给 2 个符号，平均码长为 1，如果两个符号的出现概率相差很大，编码效率将是很低的，在这些情况下，用 Huffman 编码进一步提高编码效率的方法是块编码，另一种替换方法是算术编码，它可以获得更好的编码效率。

算术编码的思想是用区域划分来表示信源输出序列，信源输出的任何一种组合与某数值范围内的一个区域一一对应，用算术方法表示这个区域，就相当于给出了一个信源输出序列的表示。如果这种算术表示能用一个二进制数表达，那么这个二进制数值是这个信源输出序列的一种编码，因为一一对应关系存在，这种编码是唯一可解码的。

下面首先通过 2 符号信源的简单情况，描述算术编码的基本原理，然后推广到一般多符号信源的情况，并给出实际可实现的算术编码器实例。

设一个信源，它有 2 个符号 a 和 b，出现概率分别是 p 和 $1-p$，设有一个基准区域，对它进行子划分，以便与信源输出序列相对应。第一步子划分，将区域划分为$[0, p)$和$[p,1)$，它们分别对应信源输出 a 和 b，每个符号对应的区域大小与它的出现概率相等；第二步，考虑信源输出第二个符号，现在分两种情况，在第一个符号是 a 的情况下，将$[0, p)$作为基区域，进一步做子划分，如果第二个符号是 a，aa 对应区域 $[0, p^2)$，如果第二个符号是 b，ab 对应区域$[p^2, p)$；在第一个符合是 b 的情况下，将$[p,1)$作为基区域做进一步子划分，第二个符号是 a，则 ba 对应区域$[p，p+p(1-p)]$，第二个符号是 b，则 bb 对应区域$[p+p(1-p),1)$，4 个区域的长度分别是 p^2，$p(1-p)$，$(1-p)p$，$(1-p)^2$，分别是符号 aa，ab，ba 和 bb 的出现概率，这样，如果信源输出 2 个符号，任何一种符号序列，一一对应一个子划分的区域，这个过程可以继续下去，直到任意的 N 个符号。

图 2.5 表示当 p=0.8 时，至多到 3 个符号输出时，符号序列与区域划分的对应关系，竖线表示[0,1]基区域，左边是符号序列，短横线是端点，右边标注是端点的值。

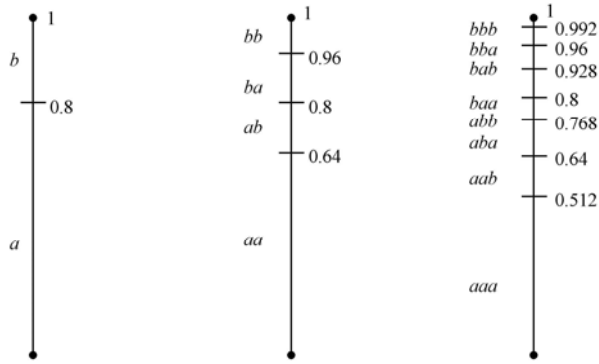


图 2.5　符号序号与区域划分示意

图 2.5 的过程继续，设输出 5 个符号 $aabaa$，按照这个划分过程，它所对应的区域为[0.512,0.593 92)，该区域端点分别用二进制表示为 [0.100 000 1…，0.100 110 0…)，可以看到，在 5 个符号输出的情况下，用二进制数 0.100 0 可以唯一地指定这个区域，因此，用码字 1 000 编码读 5 个符号的信源输出序列。

对于这个信源，它的熵 $H(X)$ =0.721 9，用直接定长编码和 Huffman 编码，其编码效率都为 72.19%，现在，算术编码用 4 比特编码 5 个符号，平均码长 R=0.8，编码效率为 90.24%，有明显提高，实际上，这只是一个很短的序列，对于长序列，理论上算术编码可以达到信源的熵。

以上讨论的工作方式，是当所有信源输出结束后，得到最后区域划分，通过该区域端点

的二进制表示，确定最终的编码，实际中，这种工作方式是不方便的，更合适的工作方式是连续编码方式：每当信源输入一个符号到算术编码器，如果可能，编码器也连续地输出编码比特流。

连续工作方式是可以实现的，通过观察图 2.5 可以发现，每增加一个新的符号，子区域划分在前一次生成的区域内部进行，因此，区域完全嵌套在前一次区域内，假设前一次区域完全包括在[0,1/2)内，后续新的子区域也总是在这个范围内，即前一次区域两个端点的二进制表示和后续所有子区域两个端点的二进制表示的第一位总是零，这一位 0 是已经确定的，可以及时地将它作为编码器输出位；同理，如果前一次区域完全包括在[1/ 2,1]内，区域两个端点的二进制表示第一位总是 1，这个 1 可以作为编码器输出位，继续推广，如果一个区域完全包括在[0,1/4），[1/ 4,1/ 2 ），[1/ 2,3/ 4 ），[3/ 4,1 ）中的任一个内，那么它的两个端点的二进制表示中有共同的 2 位，分别是 00，01，10，11，因为后续的子区域总是完全包括在这样一个区间内，这 2 位是已经确定的编码输出位。

根据以上分析，可以得到算术编码器连续工作方式为：输入 x_i,得到相应的子区域范围，确定其上下两个端点的二进制表示，获得从左边看最长相同位二进制序列，并与上一步得到的二进制序列比较，所增加的位作为编码器输出加入到输出码流，如果没有新增相同位，则这一步没有产生新的输出比特。这个过程一直重复下去，直到全部符号序列已完成。

以上讨论介绍了算数编码的基本思想，但在实际实现中，通过各种算数技巧用有限位算数运算实现算数编码仍是一件需要小心处理的事情。有多种具体实现方法，这里不再赘述，有兴趣的读者见参考文献[1]。

2.3.5　行程编码

在一些待编码的信源序列中，有些符号经常连续出现，例如，传真图像，如果按行扫描，经常出现连续的 0 或连续的多个 1，另外，在有失真编码时，信号经过变换和量化后的系数集中，经常出现连续的多个 0，对于这些情况，可以采用行程编码。

一种行程编码的形式是（*L*,*V*），如果连续出现 *V* 值 *L* 次，可以用这个偶表示。如前面提到的传真图像，只有 0 和 1 两种值，每一个值经常连续出现，可以采用这种编码形式。例如，传真图像的一个扫描行片断为 000011111100111，可以用行程编码为如下 4 个偶，（4，0）（6，1）（2，0）（3，1）。这种只有 0，1 两种符号的情况，也可以将行程编码简化为 V_0，L_1，L_2，…，L_N，其中 V_0 是起始位的值，L_1，…，L_N 为各行程的值，前面的例子可以简化为 0，4，6，2，3。

一种常用的行程编码方式为零行程，一般用于有失真编码的量化后系数编码，零是唯一经常出现的连续值，而其他值是任意的，在这种情况下，行程数 *L* 只用于表示连续 0 的个数，而偶中的数值 *V* 用于表示 0 串后紧跟的非零值，这种零行程编码也用一个偶（*L*，*V*）表示。

如 000，5，00，1，000 000，−3 用零行程编码表示为（3，5），（2，1），（6，−3）。在实际应用中，行程编码（*L*,*V*）中 *L* 和 *V* 可以分别用 Huffman 或算术编码进一步压缩其表示，在各种编码标准中，经常采用行程编码。

2.3.6　有记忆信源的编码问题

以上的讨论，主要针对无记忆信源，即相邻的信源输出之间是统计独立的，实际信源往往不满足这个条件，以下将信源条件放宽到平稳信源。

在有记忆的平稳信源情况下，$H(X)=-\sum_i P_i \mathrm{lb}P_i$ 作为 1 阶熵已经不能完全反映信源的信息量，因为信源输出序列之间存在相关性，知道前一个输出值，由于相关性，对第二个输出不可预测性就会减少。这样，为了评估有忆记信源每个输出符号平均信息量，应该从一个相当大的块出发，设块为 $X_1X_2\cdots X_N$，块的熵表示为 $H(X_1,X_2,\cdots,X_N)$，如果块的联合概率分布为 $p(x_1,x_2,\cdots x_N)$，则有：

$$H\left(X_1,X_2,\cdots,X_N\right)=-\sum_{x_1}\sum_{x_2}\cdots\sum_{x_N}p(x_1,\cdots,x_N)\mathrm{lb}p(x_1,\cdots,x_N)$$

每个符号平均熵：

$$H_N(X)=\frac{1}{N}H(X_1,X_2,\cdots,X_N)$$

当 $N\to\infty$ 时，得到平稳有记忆信源每个符号的熵 $H_\infty(X)$ 为：

$$H_\infty(X)=\lim_{N\to\infty}\frac{1}{N}H(X_1,X_2,\cdots,X_N)$$

根据联合概率分布和条件概率分布的关系得到：

$$H(X_1,X_2,\cdots,X_N)=\sum_{i=1}^{N}H(X_i\mid X_1\cdots X_{i-1})$$

进一步可以证明：

$$H_\infty(X)=\lim_{N\to\infty}H\left(X_N\mid X_1\cdots X_{N-1}\right)$$

将 $H(X)\geqslant H(X\mid Y)$ 进一步推广，得到 $H_\infty(X)\leqslant H(X)$，为了区别，在有记忆信源情况下，称 $H(X)$ 为一阶熵，由于相关性，每个符号的熵 $H_\infty(X)$ 小于一阶熵 $H(X)$，希望对有记忆信源的编码可以用更少的比特。

考虑有记忆信源的编码问题，将信源输出序列分成由 N 个连续输出符号构成的块，由编码定理知道，对块的熵编码的码长 R_{vb} 满足：

$$H(X_1,\ X_2,\ \cdots,\ X_N)\leqslant R_{\mathrm{vb}}\leqslant H(X_1,\ X_2,\ \cdots,\ X_N)+1$$

平均每个符号码长为：

$$H_N(X)\leqslant R_V<H_N(X)+1/N$$

当 $N\to\infty$ 时

$$R_V=H_\infty(X)+\varepsilon\qquad\varepsilon\to 0$$

与有记忆信源不同，在无记忆信源编码时，如果不使用块编码，由于满足：$H(X)\leqslant R_V<H(X)+1$，平均码长偏离 H（X）至多为 $0\leqslant\delta<1$，实际上在多数信源情况下，δ 更接近于 0，单符号 Huffman 编码效率一般大于 95%，在这种情况下，块编码只是将码长偏离 $H(X)$ 由 δ 变成 δ'/N（注意 δ 和 δ' 不同）。但对有记忆信源，情况不同，如果只有单符号编

码，码长为 $H(X)+\delta$，用块编码，当块充分大时，码长为 $H_\infty(X)+\delta'/N$，块编码另外一个收获是码长减少 $H(X)-H_\infty(X)$，对于相关性很强的信源，这个量可能占很大比例，块编码效率比无记忆信源时相对增加更大。

当块比较大时，块编码是比较复杂的，因为联合概率分布表变得很庞大，准确地估计也很难，一种对有记忆信源的有效编码方式是通过变换，将有记忆信源序列变换成无记忆序列，然后进行编码。

信息理论中有一个信号处理定理指出，信号通过信号处理系统，其信息是不增加的，假设一个可逆变换，表示为 T，其逆变换为 T^{-1}，那么变换和反变换都是信号处理系统，设信号 X 通过变换得到 Y，Y 通过反变换重构 X，X 是有记忆信源的输出序列，Y 是无记忆信号，那么通过 T 得到 $H(Y)\leqslant H_\infty(X)$，通过 T^{-1} 得到 $H_\infty(X)\leqslant H(Y)$，因此 $H(Y)=H_\infty(X)$，对 Y 编码，因为是无记忆的，利用单符号的 Huffman 编码或算术编码均可以获得良好效果，解码器解出 Y 后，通过 $T^{-1}Y$ 恢复 X，实际中，这个变换编码过程更多应用于有失真编码。

2.4　量　化

现实中的景物在空间和强度上都是连续量，在空间上根据采样定理对图像采样，获得一幅图像的在空间上离散、幅度上连续的样点集合，这种空间采样并不丢失信息，通过插值可以完全重构原始图像。另外，空间离散的样点集合，由于其幅度是连续的，不能用有限位的数字表示，因此，为了在计算机和其他存储媒介中存储，必须用有限的状态来逼近表示它，这就是量化过程，量化器输出的有限状态集，可以分配二进制码字来表示。

通常，量化必然产生失真，这种失真是无法恢复的，但是合理地设计量化器，使得失真尽可能小，这是量化器设计的一般准则，在信息理论中，率失真函数提供了有关量化器设计和码字表示的一个界。

在数字图像压缩应用中，量化的概念稍有不同，一个需要压缩编码的数字图像在幅度上已经是离散的，如果对图像进行变换，不管变换系数是浮点表示还是定点表示，由于数字硬件的运算精度所限，仍是离散的，因此，这里的量化过程是将“更高分辨率”的离散值量化为“更低分辨率”的离散值，为了简单和使用一般的量化理论，我们仍然将量化前高分辨率的离散值看做是连续量，而将量化器输出看做是幅度离散值。

2.4.1　率失真函数

量化引起失真，对于有失真编码，率失真函数（Rate Distortion function）给出了在限定失真不超过 D 的情况下可以达到的最低码率 R（D）。

对于一个时间离散（对图像是空间离散）、取值连续的信源，假设它是无记忆的，即相邻时刻的两个输出是统计独立的，在某时刻取值 x_n，它的量化值为 $\hat{x}_n$，其失真测量为：

$$d(x_n,\hat{x}_n)=\left|x_n-\hat{x}_n\right|^p$$

这是一个一般的失真测量，实际上常用的是 p=2，称为平方误差失真，我们在本节都采用这个测量。设信源的输出序列由 N 个样点组成，记为 X_N，平均失真测量为：

$$d(X_N,\hat{X}_N)=\frac{1}{N}\sum_{n=1}^{N}d(x_n,\hat{x}_n)$$

设信源为平稳的，最终的失真 D 定义为 $d(X_N,\hat{X}_N)$ 的统计平均：

$$D=E[d(X_N,\hat{X}_N)]=\frac{1}{N}\sum_{n=1}^{N}E[d(x_n,\hat{x}_n)]=E[d(x,\hat{x})]$$

如果每一个量化器输出都用一个二进制码表示，由于量化器输出只有有限个状态，因此，每个样点用有限位码表示，设平均码长为 R。率失真函数定义如下：对于给定的信源输出序列，如果要求失真不大于 D，即 $E[d(x,\hat{x})]\leqslant D$，每个样点所需的最小平均码长 R（D）被称为率失真函数。也就是说，在失真不大于 D 的情况下，R（D）是能够达到的最小码率。那么实际中，有没有一种编码方案能达到或任意接近 R（D）呢？香农的有关失真编码定理指出：对于给定的失真 D，总存在一种码率为 R（D）的编码方案，使得重构样本值的失真充分接近 D。

对于任意概率分布的信源，求出 R（D）的显式表达是非常困难的，但对于无记忆高斯分布的信源，R（D）的显式表达式是存在的，专门记为 R_{G}（D），并且有：

$$R_{\mathrm{G}}(D)=\begin{cases}\dfrac{1}{2}\mathrm{lb}(\sigma_x^2/D) & 0\leqslant D\leqslant\sigma_x^2\\ 0 & D<\sigma_x^2\end{cases}\qquad(2.7)$$

这个函数表示在图 2.6 中。

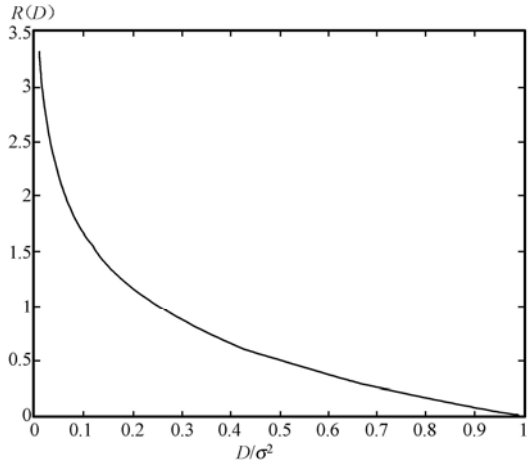


图 2.6　无记忆高斯信源的率失真函数

这里举一个数值例子，当 $D/\sigma_x^2=0.1$ 时，$R_{\mathrm{G}}(D)=1.661$，就是说，如果要求失真不大于 $D=0.1\sigma_x^2$，最好的编码方案可以达到 1.666 比特/样点的码率。

在 $0\leqslant D\leqslant\sigma_x^2$ 范围内，取式（2.7）的反函数，得率失真函数如下：

$$D_{\mathrm{G}}(R)=2^{-2R}\sigma_x^2\qquad(2.8)$$

两边取对数得：

$$10\lg D_{\mathrm{G}}(R)=-6R+10\lg\sigma_x^2$$

由此可以观察到，均方失真 D_{G}（R）按 6 dB/比特的斜率下降，每增加一比特，失真降

低 6 dB。

对于满足其他概率分布的无记忆、幅值连续的信源，它们的率失真函数的上界和下界已经被证明为 $R_{\mathrm{L}}(D)\leqslant R(D)\leqslant R_{\mathrm{G}}(D)$，这里 $R_{\mathrm{G}}(D)$是方差相等的高斯信源的率失真函数，R_{G}（D）作为上界，说明在等方差的所有分布中，高斯分布率失真函数最大，需要最多的比特进行编码。$R_{\mathrm{L}}(D)$是下界，它满足：

$$R_{\mathrm{L}}(D)=H(X)-\frac{1}{2}\mathrm{lb}2\pi\mathrm{e}D$$

其中 $H(X)$是连续无记忆信源的差分熵，对于高斯信源，由 $H_{\mathrm{G}}(X)=\frac{1}{2}\mathrm{lb}2\pi\mathrm{e}\sigma_x^2$，可以得到 $R_{\mathrm{L}}(D)=R_{\mathrm{G}}(D)$，对于其他分布的信源，$R_{\mathrm{L}}(D)<R_{\mathrm{G}}(D)$。

对于给定的率 R，失真率函数 $D(R)$也满足：

$$D_{\mathrm{L}}(R)\leqslant D(R)\leqslant D_{\mathrm{G}}(R)$$

表 2.8 给出了几种常见分布函数情况下，$H(X)$、$R_{\mathrm{G}}(D)-R_{\mathrm{L}}(D)$和 $D_{\mathrm{G}}(R)-D_{\mathrm{L}}(R)$的结果，注意，这个表给出的是上、下界，并不是准确的结果。例如，对于 Laplacian 分布，$R_{\mathrm{G}}(D)-R_{\mathrm{L}}(D)$=0.104，说明 Laplacian 分布的率失真函数最大可能比高斯的率失真函数小 0.104b，但这些结果可以启发我们，在给定的失真条件下，Laplacian 分布信源可能会比高斯分布信源得到更低的码率。

表 2.8　几种常用分布的熵和率失真性质

PDf	$P(X)$	$H(X)$	$R_{\mathrm{G}}(D)-R_{\mathrm{L}}(D)$（比特/样本）	$D_{\mathrm{G}}(R)-D_{\mathrm{L}}(R)$ /dB
高斯	$\frac{1}{\sqrt{2\pi}\sigma_x}\mathrm{e}^{-x^2/2\sigma_x^2}$	$\frac{1}{2}\mathrm{lb}(2\pi\mathrm{e}\sigma_x^2)$	0	0
均匀	$\frac{1}{2\sqrt{3}\sigma_x},\lvert x\rvert\leqslant\sqrt{3}\sigma_x$	$\frac{1}{2}\mathrm{lb}(12\sigma_x^2)$	0.255	1.53
Laplacian	$\frac{1}{\sqrt{2}ax}\mathrm{e}^{-\sqrt{2}\lvert x\rvert/\sigma_x}$	$\frac{1}{2}\mathrm{lb}(2\mathrm{e}^2\sigma_x^2)$	0.104	0.62
Gamma	$\frac{4\sqrt{3}}{\sqrt{8\pi\sigma_x\lvert x\rvert}}\mathrm{e}^{-\sqrt{3}\lvert x\rvert/2\sigma_x}$	$\frac{1}{2}\mathrm{lb}(4\pi\mathrm{e}^{0.423}\sigma_x^2/3)$	0.709	4.25

2.4.2　标量量化

仅考虑一个采样点的量化问题，这就是标量量化。设一个随机变量 x，它满足概率分布 $p(x)$，x 是量化器的输入，输出是 $\hat{x}=Q(x)$。为了设计一个量化器，用有限层逼近一个连续量，有几个参数是必须确定的。第 1 是分层数 L，第 2 是一组决策值，记为 $d_1,d_2,\cdots,d_{L-1}$，第 3 是一组代表值 $r_1,r_2,\cdots,r_L$，如果 $d_{i-1}\leqslant x<d_i$，也就是说，如果 x 取值在$[d_{i-1},d_i)$范围内，就用 r_{i}表示 x，为一般起见，设 $x\in(-\infty,+\infty)$，图 2.7 表示了决策点和代表点的关系（注意，当两个端点是有限值时，分别用 d_0 和 d_L 表示）。

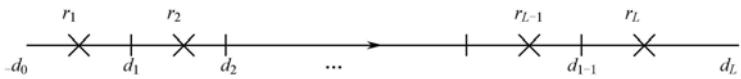


图 2.7　量化区域和代表点

常见的几种量化器类型是非均匀量化、均匀量化、带死区的均匀量化。一般定义 $\Delta_i = d_i - d_{i-1}$ 为量化步长，如果 Δ_i 随下标 i 变化，它是非均匀量化，如果 Δ_i 是常数，它是均匀量化，有一种特殊情况，当 $-\delta \leqslant x \leqslant \delta$ 时，Q（x）=0，其他情况下，Δ_i 是常数，这种情况称为带死区的均匀量化，$[-\delta, \delta]$ 称为死区，δ 是一个正值。

对于给定量化层数 L，有没有一种量化使得平均失真最小呢？答案是肯定的，Max 和 Lloyd 分别给出了均方误差意义上的最优量化器设计方法。

最优量化器：

均方意义上的量化误差表示为：

$$\begin{aligned} D &= E[(x-\hat{x})^2] = \int_{-\infty}^{+\infty} (x-\hat{x})^2 p(x)\mathrm{d}x \\ &= \sum_{k=1}^{L} \cdot \int_{d_{k-1}}^{d_k} (x-r_k)^2 p(x)\mathrm{d}x \end{aligned} \tag{2.9}$$

在已确定 L 的情况下，求解 d_k 和 r_k 使 D 最小，要求：

$$\frac{\partial D}{\partial d_k} = 0 \qquad k = 1, 2, \cdots, L-1 \tag{2.10}$$

$$\frac{\partial D}{\partial r_k} = 0 \qquad k = 1, 2, \cdots, L \tag{2.11}$$

将式（2.9）代入式（2.10）并整理得到：

$$d_k = \frac{r_k + r_{k+1}}{2} \qquad k = 1, 2, \cdots, L-1 \tag{2.12}$$

将式（2.9）代入式（2.11）并整理得到：

$$\int_{d_{k-1}}^{d_k} (x - r_k) \cdot p(x)\mathrm{d}x = 0 \qquad k = 1, \cdots, L \tag{2.13}$$

进一步得到：

$$r_k = \frac{\int_{d_{k-1}}^{d_k} x \cdot p(x)\mathrm{d}x}{\int_{d_{k-1}}^{d_k} p(x)\mathrm{d}x} \qquad k = 1, \cdots, L \tag{2.14}$$

由此可以看到，代表值 r_k 实际上是输入的概率密度函数在决策区间 $[d_{k-1}, d_k]$ 上的中心矩。

式（2.12）、式（2.14）是确定最优量化器的基本方程，但它们不是显式解，由于式（2.12）中 r_k 和 r_{k+1} 是未知的，式（2.14）中 d_k, d_{k-1} 是未知的，因此，式（2.12）和式（2.14）实际上是一组有关参数的非线性方程组，可以采用迭代的数值解法进行求解。

表 2.9 给出了高斯分布情况下，量化层分别为 2，4，8，16 时 r_k 的值，表中假设方差 $\sigma^2 = 1$，均值为 0。由于概率密度函数是对称的，量化器输出和决策层也是对称的，表中仅给出正的一半，仅给出 r_k 的值，d_k 由式（2.12）可以方便地求出，只是下标的编号需注意，下标 1 代表正的第一个代表值。

对于 L 个量化层，量化器输出用 lb$L$ 位编码，观察表 2.9 看出，每增加 1 比特，失真降

低量大于 5 dB 但小于 6 dB，这说明，最优量化加直接二进制编码，其性能达不到率失真函数 6dB/b 的下降速度，但差距小于 1 dB。

表 2.9　高斯分布的量化表

K	2 层	4 层	8 层	16 层
1	0.797 9	0.452 8	0.245 1	0.128 4
2		1.510 4	0.756 0	0.388 0
3			1.343 9	0.656 8
4			2.152 0	0.942 3
5				1.256 2
6				1.618 1
7				2.069 0
8				2.732 6
$D_{\min}$	0.363 4	0.117 5	3.455×10^{-2}	9.5×10^{-3}
$10\lg D_{\min}$	−4.396 2 dB	−9.3 dB	−14.62 dB	−20.222 8 dB

均匀量化

由于最优量化确定量化参数的过程比较复杂，在实际编码器中较少采用，而均匀量化是更简单的，量化间隔Δ一旦确定，量化器就确定了，因此，均匀量化的唯一需要确定的参数是Δ。设量化层数 L 已经确定，如果量化器输入是符合 Gaussian 或 Laplician 分布，即 $x\in(-\infty,+\infty)$，那么Δ是唯一需要确定的。初看起来，这里似乎有矛盾的要求，将区间（$-\infty,+\infty$）均匀分割成 L 层，唯一的Δ值似乎应是$+\infty$，但这是没意义的。

这里给出一个实际的均匀量化器，假设 p（x）是以 Y 轴对称的，只考虑 x>0 的一半，将 $-L/2$ 层分成内部点和端点两组，$-L/2$ −1 个内部层由宽度Δ的区间组成，决策点分别为 $0,\Delta,2\Delta,\cdots,k\Delta,\cdots,(-L/2-1)\Delta$，端点区间为 $[(-L/2-1)\Delta,+\infty]$，对于区间 $[(k-1)\Delta,\ k\Delta]$，代表点 $r_k=(2k-1)\Delta/2$，端点区间的代表点取 $(L-1)\Delta/2$，对于给定的概率密度函数 p（x）和 L，可以确定Δ，使失真最小，这个量化器的示意图如图 2.8 所示。

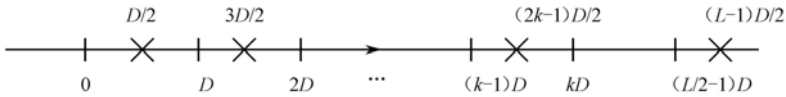


图 2.8　均匀量化器决策点和代表点示意

均匀量化的失真表示为

$$D=2\sum_{k=1}^{\frac{L}{2}-1}\int_{(k-1)\Delta}^{k\Delta}\left(\frac{2k-1}{2}\Delta-x\right)^2 p(x)\mathrm{d}x+2\int_{(\frac{L}{2}-1)\Delta}^{\infty}\left(\frac{L-1}{2}\Delta-x\right)^2 p(x)\mathrm{d}x$$

令 $\dfrac{\mathrm{d}D}{\mathrm{d}\Delta}=0$ 得：

$$\sum_{k=1}^{\frac{L}{2}-1}(2k-1)\int_{(k-1)\Delta}^{k\Delta}(\frac{2k-1}{2}\Delta-x)^2 p(x)\mathrm{d}x+(L-1)\int_{-(\frac{L}{2}-1)\Delta}^{\infty}2(\frac{L-1}{2}\Delta-x)p(x)\mathrm{d}x=0 \qquad (2.15)$$

解这个非线性方程，可以确定Δ的值，表 2.10 给出了高斯分布情况下，在给定 L 时，Δ 的最优解（方差σ^2=1）。

表 2.10　高斯信源的均匀量化步长及失真（σ^2=1）

L	最优等长Δ	最小失真 $D_{\min}$	$10\log D_{\min}$
2	1.596	0.363 4	−4.396 2
4	0.995 7	0.118 8	−9.25
8	0.586 0	3.744×10^{-2}	−14.27
16	0.335 2	1.154×10^{-2}	−19.38

比较表 2.9 和表 2.10 发现，在量化层较少，或者说是低分辨量化时，均匀量化和最优量化之间非常接近，$L\leqslant4$ 时，两种不同量化方式造成的失真的差小于或等于 0.05 dB，这几乎是可以忽略不计的，随着分层数增加，这种差距才增加。表 2.11 给出了这个比较结果，这个观察告诉我们，在低码率编码情况下，采用均匀量化无论在简单性和编码质量上，都是优选的量化方式。

很容易证明，当量化器输入符合均匀分布，且 $x\in[d_0,d_L]$ 时，最优量化和均匀量化是等价的，且有：

$$\Delta=\frac{d_L-d_0}{L}$$

$$D_{\min}=\frac{1}{12}\Delta^2$$

量化器输出+熵编码

量化器输出后是有限个符号集$\{r_1,r_2,\cdots,r_L\}$，每个符号出现的概率（记为$\{p_1,p_2,\cdots,p_L\}$）是不相同的，其中：

$$p_i=p\{Q(x)=r_i\}=\int_{d_{i-1}}^{d_i}p(x)\mathrm{d}x$$

表 2.11　均匀量化和最优量化失真比较（高斯信息 a^2=1）

L	均匀量化 $D_{\min}$	均匀量化熵	最优量化 $D_{\min}$	最优量化熵
2	−4.4	1	−4.4	1
4	−9.25	1.903 7	−9.30	1.911 1
8	−14.27	2.760 6	14.62	2.824 8
16	−19.38	3.602 4	−20.22	3.765 3
32	−24.57	×	−26.02	×
64	−29.83	×	−31.89	×
128	−35.13	×	−37.81	×

由此可以求出符号集$\{r_i,i=1,\cdots,L\}$的熵为 H_Q：

$$H_Q = -\sum_{i=1}^{L} p_i \log p_i$$

可以采用 2.3 节给出的任一种熵编码方法对 $\{r_i, i=1,\cdots,L\}$ 进行编码，进一步降低码率。

表 2.11 同时也给出了高斯信源情况下“均匀量化+熵编码”和“最优量化+熵编码”可达到的码率，从表上看到，在 L 较小时（低分辨率量化），最优量化与均匀量化性能非常接近，最优量化的失真值稍小而均匀量化的熵稍小，基本上性能是等价的。在大数目量化层时，由于最优量化比均匀量化的失真低 1～2.7 dB，直接比较它们的熵没有意义，但是已经得到的结果表明，在大间隔分层量化时，“均匀量化+熵编码”得到最优的结果，就是说，在保持失真为常数情况下，“均匀量化+熵编码”得到最小的熵，这并不限于高斯分布。

根据上述分析，在高斯或近似高斯分布情况下，在小数目分层量化（低码率）情况下，“均匀量化+熵编码”与“最优量化+熵编码”性能基本一致，由于简单性，采用均匀量化更合适。在大数目分层量化（高码率）情况下，“均匀量化+熵编码”是“标量量化+熵编码”方案中最优的，这可能是均匀量化造成的概率分布的更加不一致性（熵减小）补偿了量化部分造成的较大失真，从而可以用更多一些的分层数减小失真，而总体上在失真相同的情况下，熵仍然小于最优量化的情况。

可以得到基本结论，在标量量化情况下，如果要得到固定长编码，应采用最优量化，但如果与熵编码结合，构成变长编码，则可以采用均匀量化，实用中，为了进一步控制量化后零输出数目，常采用带死区的均匀量化。对于均匀分布，最优量化和均匀量化是一致的。目前，大多数图像与视频编码技术采用的是均匀量化或带死区的均匀量化后跟熵编码的方式。

2.5　预 测 编 码

预测和变换技术是构成当前图像编码器的最常用技术，预测和变换的主要目的是降低图像原始空间域表示中存在的非常强的相关性，使得预测或变换后的数据阵列变成低相关性（理想情况是无相关性）的稀疏阵列，这样可以用标量量化和一阶熵编码技术进行有效的压缩，因此，预测和变换的首要目的实际是降低编码器的复杂性，从而用较简单的结构设计出高效率的压缩编码算法。

在图像中，相邻像素间存在很强的相关性，由前面的像素值预测当前的像素值，由实际值减去预测值得到预测误差，强相关性使得预测值比较接近实际值，因此预测误差序列是接近零均值和具有更小方差的序列。一般预测误差的一阶熵远小于原像素值的一阶熵，对预测误差直接进行单符号的熵编码或对预测误差进行量化再进行熵编码，会取得比直接对单像素值编码更好的效率。

在预测编码方法中，最主要的方法是差分脉冲编码（Differential Pulse Code Modulation，DPCM），以上叙述的预测编码的原理实际就是 DPCM，它的编码器和解码器的基本结构如图 2.9 所示。

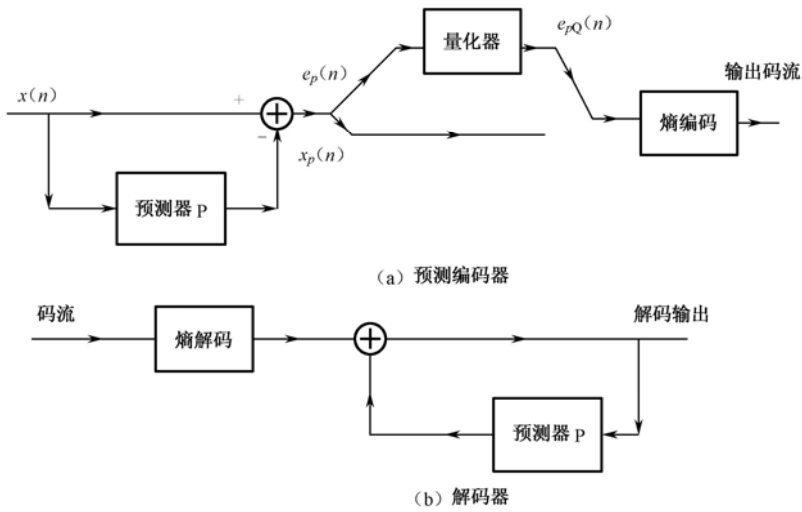


图 2.9　DPCM 的基本结构

图 2.9（a）所示的 DPCM 编码器由预测器，加法器，量化器和熵编码（变长变码 VLC）构成，预测器由过去的像素值 $x(n-1),x(n-2),\cdots x,(n-p)$ 预测当前的像素值，p 是预测器阶数，如果采用线性预测，则预测值 x_p（n）写为：

$$x_p(n)=\sum_{k=1}^{p}a_k\cdot x(n-k) \tag{2.16}$$

如果信号源是宽平稳的，信号的相关函数 $r(k)=E[x(n)\cdot x(n-k)]$ 是已知的，且信号的均值为 0（图像信号通常均值非零，可以先去均值后再处理）均方意义下的最优预测器是存在的，它是 Yule-Walker 方程的解：

$$\boldsymbol{R}\cdot\boldsymbol{A}=\boldsymbol{r} \tag{2.17}$$

这里 $\boldsymbol{A}=[a_1,a_2,\cdots,a_k]^{\mathrm{T}},\boldsymbol{r}=[r(1),r(2),\cdots,r(p)]^{\mathrm{T}}$

$$\boldsymbol{R}=\begin{bmatrix} r(0) & r(1) & \cdots & r(p-1) \\ r(1) & r(0) & r(1) & r(p-2) \\ \cdots & & & \\ r(p-1) & r(p-2) & \cdots & r(0) \end{bmatrix}$$

$\boldsymbol{R}$ 称为相关矩阵，它一般是正定的，解这个方程可以得到最优线性预测系数 $\boldsymbol{A}=\boldsymbol{R}^{-1}\boldsymbol{r}$，最小预测误差是零均值，方差为：

$$\sigma_e^2=\sigma_x^2-r^H\cdot\boldsymbol{A}=\sigma_x^2-\sum_{k=1}^{P}a_k\cdot r(k) \tag{2.18}$$

一个一般线性预测器的结构如图 2.10 所示，图中 z^{-1} 为延迟单元。

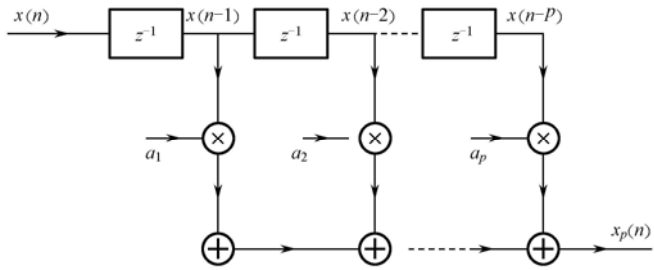


图 2.10　线性预测器结构

在图像编码中，前面的像素不一定是同一行的像素，也可能是上一行的像素，对于电视图像，还可能是前一场或前一帧的像素，暂不考虑电视图像问题，图 2.11 表示了一个常用的“前面像素”的选择，x 代表当前像素，等价为 x（n），A、B、C、D 为前面像素，等价为 x（n−1）,⋯,x（n−4），可以通过这些像素位置之间的相关函数确定最优预测系数。

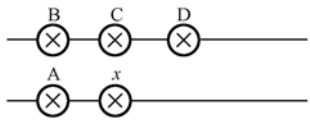


图 2.11　用 A、B、C、D 预测 x

当预测器系数确定以后，每一个新的像素 x（n），得到一个预测误差 e_p（n），如果进行无失真压缩，这个预测误差 e_p（n）直接进行熵编码，得到输出码流。如果要进行有失真压缩，对 e_p（n）进行量化后再实行熵编码，解码器是一个相反的过程，同样参数的预测器设置在解码器中，由前面已解码值预测当前值，预测值与经熵解码的解码预测误差相加重构当前值。

注意，图 2.9 的结构是一个原理性的说明，或仅适用于无失真预测编码的情况，在有失真编码时，会引起误差积累，在连续几个像素后，因误差积累造成解码器输出远远偏离实际值 x（n），造成这个问题的原因是在编码器和解码器的内部，两个预测器使用了不同的输入，在编码器，直接由 $\{x(n-1),\ x(n-2),\ \cdots,\ x(n-k)\}$ 预测 $x(n)$，但在解码器，仅有重构的 $\{\hat{x}(n-1),\hat{x}(n-2),\cdots,\hat{x}(n-k)\}$ 预测 $x(n)$，由于有失真编码器的重构值与编码前的真实值有误差，因此，解码器的预测值 $\hat{x}_p(n)$ 不等于真正的预测值 $x_p(n)$，这就造成了附加失真，我们进一步定量看一下解码器造成的附加失真。

在编码器，预测值是：

$$x_p(n)=\sum_{K=1}^{p}a_k\cdot x(n-k)$$

预测误差是：

$$e_p(n)=x(n)-x_p(n)$$

$e_p(n)$ 的量化值 $e_{pq}(n)=Q(e_p(n))$，量化误差为 $q(n)=e_{pq}(n)-e_p(n)$，如果解码器不存在附加误差，那么解码器的输出为：

$$\hat{x}(n)=x_p(n)+e_{pq}(n)=x(n)+q(n)$$

解码器输出与原像素之间的误差仅为量化误差 q（n）。

观察图 2.9 的情况，图中的解码器的实际预测输出为（为简单计，假设在此之前没有附加误差）：

$$\begin{aligned}\hat{x}_p(n) &= \sum_{k=1}^{p} a_k \cdot \hat{x}(n-k) \\ &= \sum_{k=1}^{p} a_k [x(n-k) + q(n-k)] \\ &= \sum_{k=1}^{p} a_k x(n-k) + \sum_{k=1}^{p} a_k \cdot q(n-k) \\ &= x_p(n) + \delta\end{aligned}$$

实际重构值：

$$\begin{aligned}\hat{x}(n) &= \hat{x}_p(n) + e_{pq}(n) = x_p(n) + \delta + e_{pq}(n) \\ &= x(n) + q(n) + \delta\end{aligned}$$

图 2.9 的解码器引起重构值有一个 δ 的附加失真，由于 $\hat{x}(n)$ 还会进一步用于未来像素解码的预测器输入，这种误差是进一步积累的，引起重构图像产生严重偏离，解决这个问题的方法是改进编码器，使编码器和解码器的预测器使用相同的输入，图 2.12 是一个改进后的实际预测编码器和解码器结构。

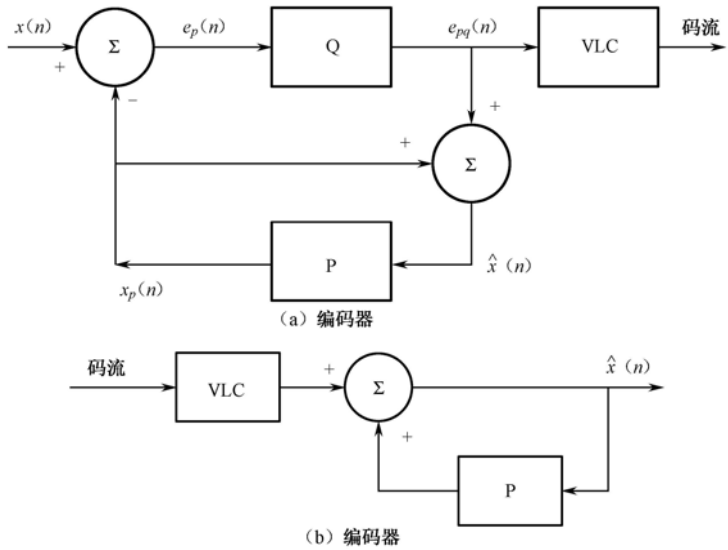


图 2.12　改进后的实际预测编/解码器结构

在图 2.12 中，编码器和解码器使用相同的输入集 $\{\hat{x}(n-1),\cdots,\hat{x}(n-p)\}$，注意到，编码器中，实际上内嵌了一个解码器（除变长解码 VLD 外），它得到了与解码器相同的重构值 $\hat{x}(n)$，并将它输入到预测器的延迟/保存单元之中。

在许多实际编码器中，量化器 Q 的输出一般不是直接的输入的逼近值（或称为代表值），而是代表值的序号，因此，解码器中一般需要一个反量化器 Q^{-1}，它的功能是输入一个序号，通过内部计算或查表，输出一个代表值，考虑到这种实际量化器和反量化器的功能，图 2.12

的结构进一步改进为图 2.13。

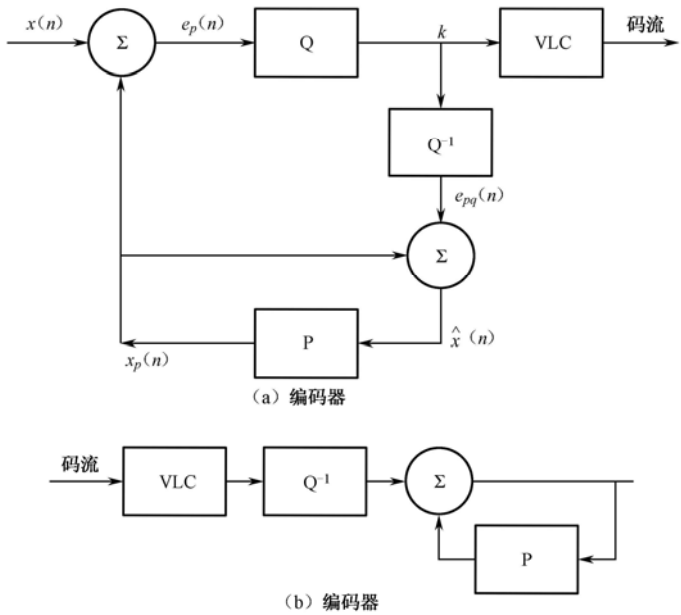


图 2.13　一般 DPCM 的结构

关于电视图像的帧间预测和运动补偿，专门在第 2.7 节讨论，此处不再赘述。

2.6　变 换 编 码

变换的应用更加广泛，理想的变换不仅去除相关，还可以将信号的主要能量集中在很少的几个系数上，这实际是信号的一种紧致表示，通过量化将细节信号去除，使量化系数矩阵变成一个非常稀疏的矩阵，用很少的量化系数逼近原始信号，然后通过有效的表示量化系数中的非零值位置和幅度得到高压缩率。

变换编码是目前应用最广泛的图像压缩编码方法，几乎所有的图像（除 2 值图像外）和视频压缩标准均以变换编码为主要工具。变换编码在压缩比，重构图像质量，适应范围和算法复杂性多方面获得好的拆中，在实际上得到广泛应用。

变换编码的基础：图像变换、快速算法、量化理论与人类视觉系统特性的结合（HVS）、扫描技术、变长编码等，都被广泛地研究，发表了大量研究结果，本节只给出一个简要的说明，介绍变换编码涉及的基本理论、算法和编码器结构，重点放在 DCT 变换编码方法上，它是目前使用最广泛的技术。

2.6.1　一般图像变换

在本节中，为了讨论方便，我们使用几个约定的符号。

一个二维图像用 x（i，j）表示，其中，$0 \leqslant i < M$，$0 \leqslant j < N$，像素集构成的矩阵记为 $\boldsymbol{X} = [x(i,j)]_{M \times N}$，对图像进行变换，图像的变换域表示为 y（k,l），变换系数矩阵表示为

$\boldsymbol{Y}=[y(k,l)]_{M\times N}$。对于一个一维矢量$\boldsymbol{U}=[u(0),u(1),\cdots,u(N-1)]^{\mathrm{T}}$，对 $\boldsymbol{U}$ 进行变换得到矢量$\boldsymbol{V}=[v(0),v(1),\cdots,v(N-1)]^{\mathrm{T}}$。

首先简要叙述一维变换情况，然后推广到二维变换。

一个一维线性变换的一般形式表示为：

$$\boldsymbol{V}=\boldsymbol{A}\cdot\boldsymbol{U} \tag{2.19}$$

这里 $\boldsymbol{A}$ 是变换矩阵，$\boldsymbol{V}$ 和 $\boldsymbol{U}$ 是 $\boldsymbol{N}$ 维矢量，$\boldsymbol{U}$ 是原始矢量，$\boldsymbol{V}$ 是变换系数矢量，如果变换矩阵 $\boldsymbol{A}$ 满足：

$$\boldsymbol{A}\cdot\boldsymbol{A}^{\mathrm{H}}=\boldsymbol{I} \tag{2.20}$$

这里上标 H 表示共轭转置，这个变换称为酉变换，反变换为：

$$\boldsymbol{U}=\boldsymbol{A}^{\mathrm{H}}\cdot\boldsymbol{V} \tag{2.21}$$

由反变换可以完全重构原矢量。

如果矩阵 $\boldsymbol{A}$ 的第 k 行，n 列的元素记为 a（k,n）=a_k（n），线性变换和反变换也可以写成常见的求和形式：

$$v(k)=\sum_{n=0}^{N-1}a_k(n)\cdot u(n)\qquad 0\leqslant k\leqslant N-1 \tag{2.22}$$

$$u(n)=\sum_{k=0}^{N-1}a_k^*(n)\cdot v(k)\qquad 0\leqslant n\leqslant N-1 \tag{2.23}$$

为了更清楚地了解正交变换的物理意义，记$\boldsymbol{A}^{\mathrm{H}}$的第$k$行为$\boldsymbol{a}_k^*$，$\boldsymbol{a}_k^*=[a_k^*(0),a_k^*(1),\cdots,a_k^*(N-1)]^{\mathrm{T}}$，式（2.23）中 N 个求和项可以表示成矢量形式：

$$\boldsymbol{U}=\begin{bmatrix}u(1)\\u(2)\\\vdots\\u(N-1)\end{bmatrix}=\sum_{k=0}^{N-1}v(k)\cdot\vec{\boldsymbol{a}}_k^*=\sum_{k=0}^{N-1}v(k)\cdot\begin{bmatrix}a_k^*(0)\\a_k^*(1)\\\vdots\\a_k^*(N-1)\end{bmatrix} \tag{2.24}$$

这里 $v(k)$表示为：

$$v(k)=\sum_{n=0}^{N-1}a_k(n)\cdot u(n)=\boldsymbol{U}\cdot\boldsymbol{a}_k^{*\mathrm{T}}=\left\langle\boldsymbol{U},\boldsymbol{a}_k^*\right\rangle \tag{2.25}$$

式（2.25）中 $v(k)$是矢量 $\boldsymbol{U}$ 在矢量$\boldsymbol{a}_{\mathrm{k}}^*$上的投影，式（2.24）将矢量 $\boldsymbol{U}$ 分解为 N 个矢量$\{\boldsymbol{a}_1^*,\boldsymbol{a}_2^*,\cdots,\boldsymbol{a}_{N-1}^*\}$的线性组合，组合系数 v（k）表示 $\boldsymbol{U}$ 中包含$\boldsymbol{a}_{\mathrm{k}}^*$矢量的强度，正是这样，$\boldsymbol{a}_{\mathrm{k}}^*$称为变换的基函数，矢量 $\boldsymbol{V}$ 称为变换系数矢量。

例如，$a_k(n)=\mathrm{e}^{j\frac{2\pi}{N}k\cdot n}$

矢量$\boldsymbol{a}_k=[1,\mathrm{e}^{j\frac{2\pi}{N}k},\mathrm{e}^{j\frac{2\pi}{N}k\cdot 2},\cdots,\mathrm{e}^{j\frac{2\pi}{N}k\cdot(N-1)}]^{\mathrm{T}}$表示角频率为$\frac{2\pi}{N}k$的矢量，$v$（$k$）表示 $\boldsymbol{U}$ 中包括的频率$\frac{2\pi}{N}k$分量复正弦的成分，其中 v（0）代表直流分量的成分，这实际是读者熟知的离散傅里叶变换（DFT）。

最后实际上是式（2.20）的求和形式，写出酉变换的正交性和完整性条件：

$$\sum_{n=0}^{N-1} a_k(n)\cdot a_{k'}(n)=\delta(k-k') \tag{2.26}$$

$$\sum_{k=0}^{N-1} a_k(n)\cdot a_k(n')=\delta(n-n') \tag{2.27}$$

将以上结果推广到二维的情况，二维变换的求和表示为：

$$y(k,l)=\sum_{m=0}^{M-1}\sum_{n=0}^{N-1} x(m,n)\cdot a_{k,l}(m,n)\quad 0\leqslant k\leqslant M-1, 0\leqslant l\leqslant N-1 \tag{2.28}$$

$$x(m,n)=\sum_{k=0}^{M-1}\sum_{l=0}^{N-1} y(k,l)\cdot a^*_{k,l}(m,n)\quad 0\leqslant m\leqslant M-1, 0\leqslant n\leqslant N-1 \tag{2.29}$$

下面为讨论简单，设 M=N，这个二维变换需要 N^4 的运算，如果变换核函数 $a_{k,l}(m,n)$ 是可分的，那么变换是可分的，运算量降到 $2N^3$，所谓可分是指：

$$a_{k,l}(m,n)=a_k(m)\cdot a_l(n) \tag{2.30}$$

例如，$a_{k,l}(m,n)=\mathrm{e}^{j\frac{2\pi}{N}(km+\ln)}=\mathrm{e}^{j\frac{2\pi}{N}km}\mathrm{e}^{j\frac{2\pi}{N}\ln}=a_k(m)\cdot a_l(n)$

由可分性，式（2.28），式（2.29）可以写成为：

$$y(k,l)=\sum_{m=0}^{N-1}\sum_{n=0}^{N-1}\cdot a_k(m)\cdot x(m,n)\cdot a_l(n) \tag{2.31}$$

$$x(m,n)=\sum_{k=0}^{N-1}\sum_{l=0}^{N-1} a^*_k(m)\cdot y(k,l)\cdot a^*_l(n) \tag{2.32}$$

或写成矩阵形式：

$$\boldsymbol{Y}=\boldsymbol{A}\cdot\boldsymbol{X}\cdot\boldsymbol{A}^{\mathrm{T}} \tag{2.33}$$

$$\boldsymbol{X}=\boldsymbol{A}^{\mathrm{H}}\cdot\boldsymbol{Y}\cdot\boldsymbol{X}^* \tag{2.34}$$

式（2.33）可以写成：

$$\boldsymbol{Y}^{\mathrm{T}}=\boldsymbol{A}\cdot(\boldsymbol{A}\cdot\boldsymbol{X})^{\mathrm{T}} \tag{2.35}$$

或

$$\boldsymbol{Y}=\boldsymbol{A}\cdot(\boldsymbol{A}\cdot\boldsymbol{X}^{\mathrm{T}})^{\mathrm{T}} \tag{2.36}$$

式（2.35）是先对 $\boldsymbol{X}$ 的每一列做变换，对变换完的矩阵，再对每一行做变换，式（2.36）是先对 $\boldsymbol{X}$ 的每一行做变换，再对列做变换，可分的二维变换可以由对每一行（或列）做变换，再对每一列（或行）做变换这样两遍共 $2N$ 个一维变换完成。

与一维变换的基函数 $\{\boldsymbol{a}^*_k, k=0,1,\cdots N-1\}$ 对应，二维变换存在基图像（或称为二维基函数），不难理解，一个可分的二维变换的基图像 $\boldsymbol{A}^*_{k,l}$ 定义为：

$$\boldsymbol{A}^*_{k,l}=\boldsymbol{a}^*_k\cdot\boldsymbol{a}^{*\mathrm{T}}_l \tag{2.37}$$

容易检证：

$$X=\sum_{k=0}^{N-1}\sum_{l=0}^{N-1} y(k,l)\cdot A^*_{k,l} \tag{2.38}$$

$$y(k,l)=\left\langle X, A^*_{k,l}\right\rangle \tag{2.39}$$

式（2.38）、式（2.39）的解释与式（2.14）、式（2.15）一致，现在基图像 $\boldsymbol{A}^*_{k,l}$ 是 $N\times N$

的子图像，这样的子图像共 $N\times N$ 个，选择不同的基图像，得到不同的物理解释，例如，二维 DFT，基图像 $\boldsymbol{A}_{k,l}^{*}$ 表示二维复正弦分量，k 表示垂直频率，l 表示水平频率，$\boldsymbol{A}_{0,0}^{*}$ 表示直流图像，y（0,0）表示 X 中含的直流分量。

对于另外一个重要变换，即离散余弦变换（Discrete Cosine Transform，DCT）：

$$a_k(n)=\begin{cases}\dfrac{1}{\sqrt{N}} & k=0\\ \sqrt{\dfrac{2}{N}}\cos\dfrac{\pi(2n+1)k}{2N} & 1\leqslant k\leqslant N-1\end{cases} \tag{2.40}$$

取 N=8 时，它的 8×8 个基图像如图 2.14 所示，每一个 $\boldsymbol{A}_{k,l}^{*}$ 表示一种模式，$\boldsymbol{A}_{0,0}^{*}$ 是直流，$\boldsymbol{A}_{N-1,0}^{*}$ 是垂直高频、水平低频模式，$\boldsymbol{A}_{0,N-1}^{*}$ 是垂直低频、水平高频模式，$\boldsymbol{A}_{N-1,N-1}^{*}$ 是垂直和水平均为高频模式，因此，如果进行二维 DCT 变换，y（0,0）是直流分量，y（N−1,0）是垂直高频，y（0,N−1）是水平高频分量，y（N−1,N−1）是两方向最高频分量，其他系数处于这些频率之间。

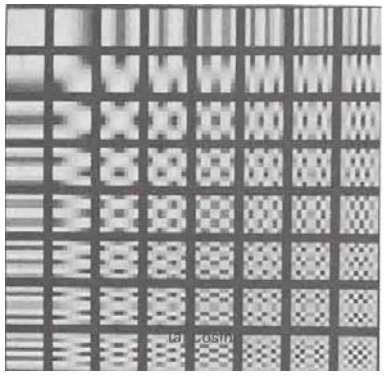


图 2.14　二维 DCT 变换的基图像

酉变换是能量保持的，它有一个重要性质性质，即满足：

$$\sum_{m=0}^{N-1}\sum_{n=0}^{N-1}\left|x(m,n)\right|^2=\sum_{K=0}^{N-1}\sum_{l=0}^{N-1}\left|y(k,l)\right|^2 \tag{2.41}$$

如果变换域内有失真，得到 $\hat{y}(k,l)$，相应空间域内失真也存在，记为 $\hat{x}(m,n)$，则误差能量为：

$$e=\sum_{m=0}^{N-1}\sum_{n=0}^{N-1}\left|x(m,n)-\hat{x}(m,n)\right|^2=\sum_{k=0}^{N-1}\cdot\sum_{l=0}^{N-1}\left|y(k,l)-\hat{y}(k,l)\right|^2 \tag{2.42}$$

有许多酉变换（复变换核）和线性正交变换（实变换核）的例子，由于本节主要讨论 DCT 变换，这些变换核仅在此简单讨论。

DFT

正如前面所给出的，它的变换核函数为：

$$a_k(n)=\mathrm{e}^{j\frac{2\pi}{N}nk}$$

1）Hadamard 变换

设变换序列 $N=2^n$，Hadamard 变换矩阵 $\boldsymbol{H}_n$ 写成如下递推关系：

$$\boldsymbol{H}_n=\frac{1}{\sqrt{2}}\begin{pmatrix}\boldsymbol{H}_{n-1}, & \boldsymbol{H}_{n-1}\\ \boldsymbol{H}_{n-1}, & -\boldsymbol{H}_{n-1}\end{pmatrix}$$

由初始值 $\boldsymbol{H}_1=\frac{1}{\sqrt{2}}\begin{pmatrix}1 & 1\\ 1 & -1\end{pmatrix}$ 递推可以得到长度为 8 的变换矩阵为：

$$\boldsymbol{H}_3=\frac{1}{\sqrt{8}}\begin{bmatrix}1 & 1 & 1 & 1 & 1 & 1 & 1 & 1\\ 1 & -1 & 1 & -1 & 1 & -1 & 1 & -1\\ 1 & 1 & -1 & -1 & 1 & 1 & -1 & -1\\ 1 & -1 & -1 & 1 & 1 & -1 & -1 & 1\\ 1 & 1 & 1 & 1 & -1 & -1 & -1 & -1\\ 1 & -1 & 1 & -1 & -1 & 1 & -1 & 1\\ 1 & 1 & -1 & -1 & -1 & -1 & 1 & 1\\ 1 & -1 & -1 & 1 & -1 & 1 & 1 & -1\end{bmatrix}$$

2）Haar 变换

Haar 变换相当于用两个滤波器 $\boldsymbol{H}_l=\left\{\frac{1}{\sqrt{2}},\frac{1}{\sqrt{2}}\right\}$ 和 $\boldsymbol{H}_h=\left\{\frac{1}{\sqrt{2}},\frac{-1}{\sqrt{2}}\right\}$，分别对数据序列滤波，并进行 2∶1 亚采样，得到的高频输出保持，而低频部分继续这样的分解过程，直到最后只有一个点，设 $N=2^n$，分解过程需进行 n 次，这个过程也等价为一个线性变换。$N=8$ 的 Haar 变换矩阵为：

$$\boldsymbol{H}_{r3}=\frac{1}{\sqrt{8}}\begin{bmatrix}1 & 1 & 1 & 1 & 1 & 1 & 1 & 1\\ 1 & 1 & 1 & 1 & -1 & -1 & -1 & -1\\ \sqrt{2} & \sqrt{2} & -\sqrt{2} & -\sqrt{2} & 0 & 0 & 0 & 0\\ 0 & 0 & 0 & 0 & \sqrt{2} & \sqrt{2} & -\sqrt{2} & -\sqrt{2}\\ 2 & -2 & 0 & 0 & 0 & 0 & 0 & 0\\ 0 & 0 & 2 & -2 & 0 & 0 & 0 & 0\\ 0 & 0 & 0 & 0 & 2 & -2 & 0 & 0\\ 0 & 0 & 0 & 0 & 0 & 0 & 2 & -2\end{bmatrix}$$

3）Slant 变换

Slant 变换的核函数，也可以通过递推公式求出，此处从略，仅给出一个 $N=2^2$ 的变换矩阵的例子：

$$\boldsymbol{S}_2=\frac{1}{2}\begin{bmatrix}1 & 1 & 1 & 1\\ \frac{3}{\sqrt{5}} & \frac{1}{\sqrt{5}} & \frac{-1}{\sqrt{5}} & \frac{-3}{\sqrt{5}}\\ 1 & -1 & -1 & 1\\ \frac{1}{\sqrt{5}} & \frac{-3}{\sqrt{5}} & \frac{3}{\sqrt{5}} & \frac{-1}{\sqrt{5}}\end{bmatrix}$$

2.6.2　DCT 变换

在图像变换中，具有最好的去相关和能量紧致特性的变换是 KL 变换，但 KL 变换的变换矩阵是依赖于具体图像的，对于一幅给定图像，需要估计它的相关矩阵，然后进行特征值分解，得到特征矢量，才能得到变换矩阵，由于变换矩阵不确定，KL 变换也没有通用的高效的快速算法，这些都使得 KL 变换在实际应用中是很复杂的。

离散余弦变换（DCT）被认为是在高相关性的随机矢量情况下对 KL 变换的很好逼近，因此，DCT 广泛应用于图像压缩中，一维 DCT 变换的核函数是：

$$a_k(n)=\begin{cases}\dfrac{1}{\sqrt{N}} & k=0,0\leqslant n\leqslant N-1\\ \sqrt{\dfrac{2}{N}}\cos\dfrac{\pi(2n+1)k}{2N} & 1\leqslant k\leqslant N-1,0\leqslant n\leqslant N-1\end{cases}\tag{2.43}$$

因此，矢量 $\boldsymbol{U}=\left[u(0),u(1),\cdots,u(N-1)\right]^{\mathrm{T}}$ 的一维 DCT 为：

$$v(k)=\partial(k)\cdot\frac{1}{\sqrt{N}}\sum_{n=0}^{N-1}u(n)\cos\frac{\pi(2n+1)k}{2N},\quad 0\leqslant k\leqslant N-1\tag{2.44}$$

这里 $\begin{cases}\partial(0)=1\\ \partial(k)=\sqrt{2} & 1\leqslant k\leqslant N-1\end{cases}$

反变换（IDCT）为：

$$u(n)=\frac{1}{\sqrt{N}}\sum_{k=0}^{N-1}\partial(k)\cdot v(k)\cdot\cos\frac{\pi(2n+1)k}{2N}\tag{2.45}$$

DCT 同样可以表示成矩阵形式，设变换矩阵为 $\boldsymbol{C}$，$\boldsymbol{C}$ 中第 k 行，第 n 列元素 $C(k,n)=a_k(n)$，DCT 变换矩阵形式为：

$$\boldsymbol{V}=\boldsymbol{C}\cdot\boldsymbol{U}$$

反变换为：

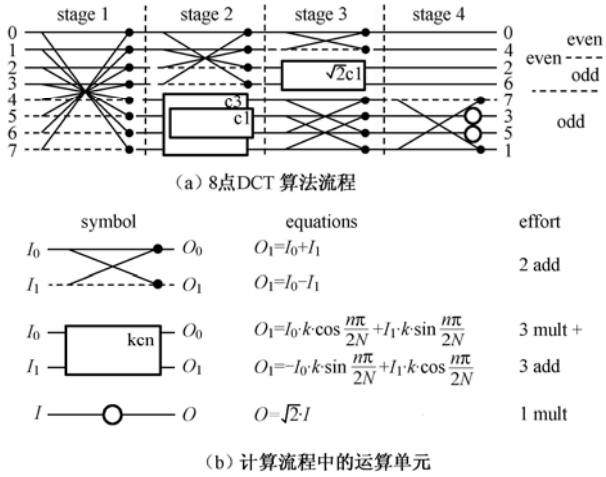
$$\boldsymbol{U}=\boldsymbol{C}^{\mathrm{H}}\cdot\boldsymbol{V}$$

DCT 变换的另一个优点是存在高效的快速算法，快速 DCT 有很多不同类型的算法，利用 $2N$ 点或 N 点 FFT 可以引出快速 DCT 算法，但更有效的还是利用 DCT 的性质构造专用的快速算法，这方面结果很多，这里仅简要介绍 Loeffle 等提出的一个快速算法，它的结构规范，适用于硬件实现，对于 8 点 DCT 仅需要 11 次乘法和 29 次加法，乘法次数达到理论上的下限，这个算法的流程图如图 2.15（a）所示，其中图 2.15（a）中的主要运算单元示于图 2.15（b），注意，在图 2.15（b）的运算单元中，用下列等式可以将 4 次乘法、2 次加法运算变成

3 次乘法和 3 次加法：

$$y_0 = ax_0 + bx_1 = (b-a)x_1 + a(x_0 + x_1)$$
$$y_1 = -bx_0 + ax_1 = -(a+b)x_0 + a(x_0 + x_1) \tag{2.46}$$

由于 a，b 是常数，$a+b$，$b-a$ 可以预先算好存在寄存器中，不需要增加运算，对于快速反变换，只需要改变输入、输出方向。



（a）8点DCT 算法流程

（b）计算流程中的运算单元

图 2.15　8 点 DCT 的快速算法

16 点 DCT 的快速算法如图 2.16 所示。

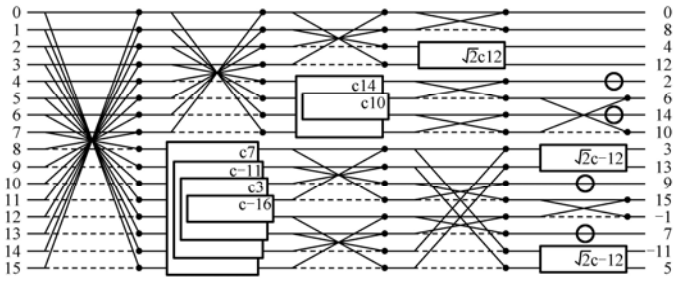


图 2.16　16 点 DCT 的快速算法

二维 DCT 变换定义如下：

$$y(k,l) = \partial(k)\partial(l)\frac{1}{\sqrt{N\cdot M}}\sum_{m=0}^{M-1}\sum_{n=0}^{N-1}x(m,n)\cdot\cos\frac{\pi(2m+1)k}{2M}\cdot\cos\frac{\pi(2n+1)l}{2N} \tag{2.47}$$

反变换为：

$$x(m,n) = \frac{1}{\sqrt{N\cdot M}}\sum_{k=0}^{M-1}\sum_{l=0}^{N-1}\partial(k)\cdot\partial(l)\cdot\cos\frac{\pi(2m+1)k}{2M}\cdot\cos\frac{\pi(2n+1)l}{2N} \tag{2.48}$$

由于二维 DCT 变换是可分的，一维快速 DCT 算法可以用于高效计算二维 DCT 变换，尽管也存在直接计算二维 DCT 的快速算法，本节不再赘述。

在本节结束前，通过一个数学模型和一个实际图像，进一步说明 DCT 变换在能量紧凑特性上逼近最优变换 KLT，而优于其他变换。

关于数字模型，是取一个长度为 16 的一阶 Markov 平稳序列，ρ=0.95，用前 m 个变换系数近似表示原序列，均方归一化剩余误差定义为：

$$J_m = \frac{\sum_{k=m}^{N-1} a_k^2}{\sum_{k=0}^{N-1} a_k^2} \tag{2.49}$$

这个值越小，说明能量紧凑特性越好，图 2.17 给出了多种变换的性能比较，从图中看出，对于这个模型，DCT 变换与 KLT 变换性能基本一致，好于其他变换。

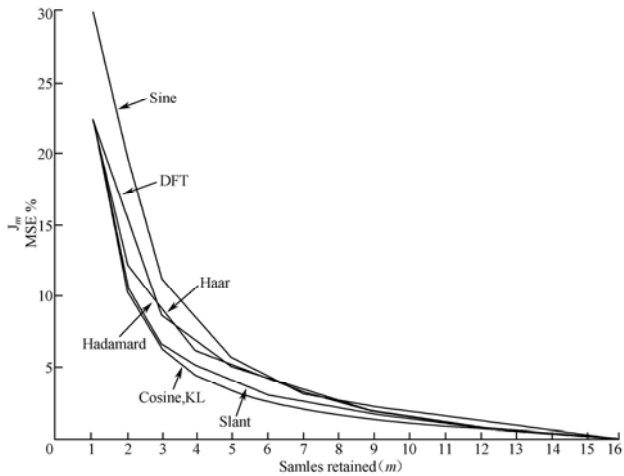


图 2.17　各种变换能量紧凑特性比较

2.6.3　变换编码

在信息理论中，有一个信号处理定理说，一个信号通过信号处理系统，它的熵不会增，由于线性正交变换和反变换都可以认为是一个信号处理系统，并且满足理想重构条件，因此，图像进行变换后，熵不会变化，变换的目的不是减小熵，而是使编码器得到简化。图像空间域存在很强的相关性，并且较远距离像素间（例如，相隔 10 行，10 列）相关系数仍较大，这要求空间域直接编码必须采用大维数的矢量结构，通过变换，大大减小了系数之间的相关性，如果是采用 KLT，平稳信号的变换系数完全不相关。实际图像编码中最常采用的 DCT，由于是 KLT 的非常近似的逼近，使 DCT 系数之间是近似不相关的，这个不相关性可以简化编码器的设计，使得标量量化（仅存在于有失真压缩）结合熵编码可以获得接近于矢量处理的结果。

变换不改变信号的熵，但却可以改变一阶熵，使编码器简化；另外，在有失真编码时，变换改变率失真函数。在保持一定的失真条件下，KLT 可以达到最小的率，DCT 可以逼近这个率，通过选择 DCT 变换，在正交变换中，可以得到近似最高的压缩率，这也是变换编码的另一个好处。

在 DCT 为主要方法的变换编码中，一般不直接对整个图像进行变换，而是首先对图像分块，将 $M\times N$ 的一幅图像，分成不重叠的 $\frac{M}{k}\times\frac{N}{k}$ 个 $k\times k$ 块分别进行变换，这样做，从总体效率-失真性能上不会带来什么收益，但却是一种性能下降很小，但实现方便性大大增强的技术。首先，从运算量上，一个 $N\times N$ 图像采用整体 DCT 变换，使用快速算法，需要的运算量平均为 $MN\text{lb}(MN)$，分块变换后，运算量降为 $2MN\text{lb}k$，对一幅 512×512 图像，分块变换仅需约 1/3 的运算量，其次，后续的量化和扫描处理可以得到明显的简化，第三是容易将传输误差引起的错误控制在一个块内，而不是在整幅图像上蔓延。采用多大的块，是一个需要回答的问题。

研究在不同块大小情况下，能量紧凑特性，可以用于指导选择块大小。通过对一个模型和一些实际图像进行测试，可以得到一些有指导意义的结果。首先选择一个模型，它是一阶 Markov 过程，保留其各阶块变换的低 1/4 变换系数，评价它的误差性能，图 2.18 给出各种变换在ρ=0.95 时各种块大小情况的结果，从图上看，KLT 和 DCT 在各种块大小的情况下好于其他变换，在块大小为 8×8 以后，增加块大小误差的下降变得平缓，从性能和简单性角度折中 8×8 和 16×16 块大小是合适的。对于实际图像 lena 和 baboon 做了类似实验，但仅对 DCT 变换进行，结果示于图 2.19，图中对 $2^n\times2^n$ 块大小进行实验，横坐标表示 $2^n\times2^n$ 块大小的指数 n。“+”是图像 baboon 的结果，“*”是图像 lena 的结果，从 n=1 到 n=7，同样，$n\geqslant3$ 开始，曲线变缓，说明 8×8 和 16×16 是合适的。

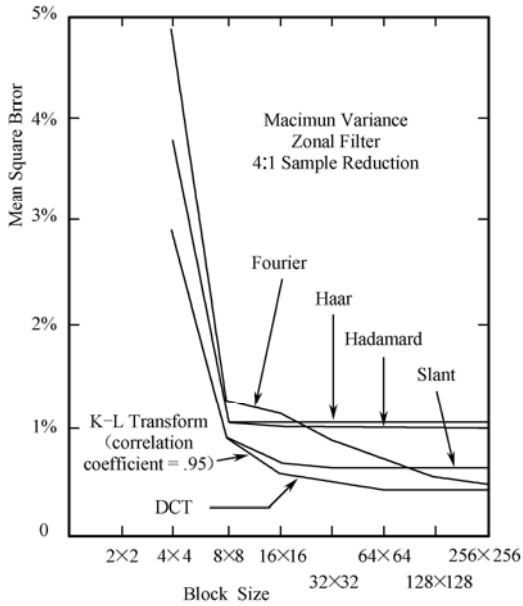


图 2.18　块变换大小对能量紧凑性影响

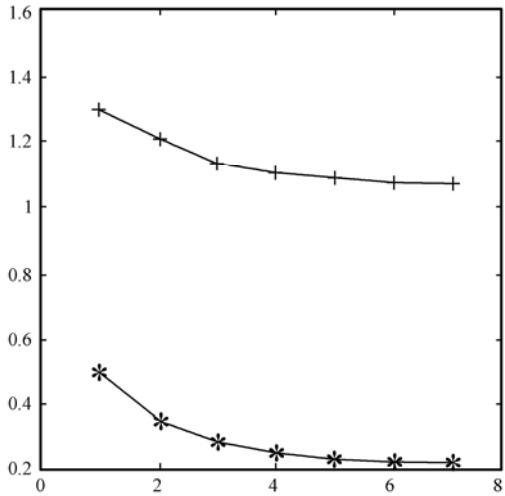


图 2.19　对实际图像块变换大小实验

采用 $k\times k$ 块 DCT 变换（k=8 或 16）结合标量量化的一般变换编码方案如图 2.20 所示，也可以用矢量量化取代最后两个单元，结合量化在内的编码器都是有失真的。

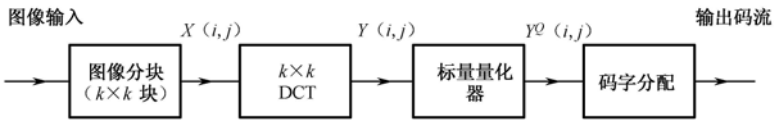


图 2.20　DCT 变换编码器

在如上讨论中，前 2 个单元已经是确定的，根据后 2 个单元的不同设计方案，可以得到不同的实际编码器，以下简单介绍几个方法，一个最常用的实际变换编码器是第 3 章介绍的 JPEG 编码标准中的基本编码模式。

2.6.4　基于 HVS 的量化与码率分配

变换编码可以灵活地结合 HVS 特性，人们已经注意到，HVS 对不同的频率分量具有不同的敏感性，将这些频率敏感性结合到变换系数，通过频率敏感性加权失真测量或设置依赖频率变化的量化步长矩阵，均可以改善压缩图像的视觉质量。

通过用不同空间频率的光栅图像，测试 HVS 的敏感性响应，得到 HVS 对不同频率成分的敏感性曲线，HVS 的调制传输函数（Modulation Transfer Function，MTF）很好地表示了这个特性。在不同的观察条件下，MTF 有所不同，图 2.21 给出几个不同观测条件下的 MTF 曲线。

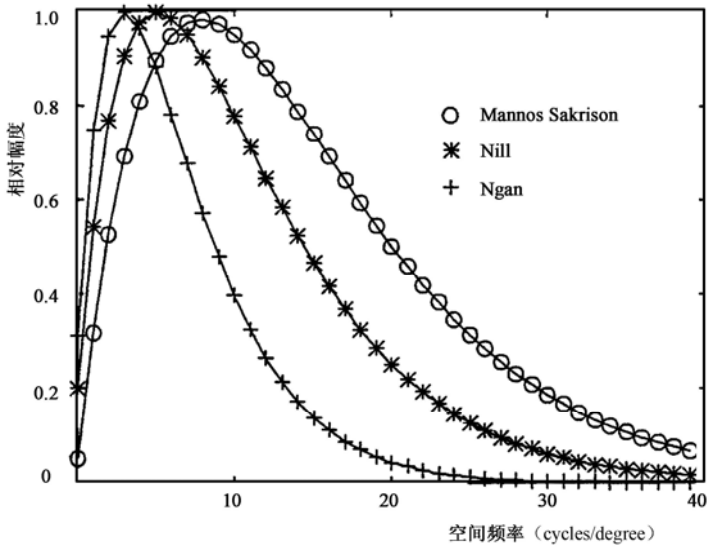


图 2.21　HVS 的 MTF 曲线

在 $K\times K$ DCT 变换情况下，在恰当的实验条件下，由 MTF 曲线可以得到代表不同频率成分变换系数 $y(k,l)$ 的加权系数 $w(k,l)$，它是由二维 MTF 曲面对 DCT 变换的频率区域划分后，在各频率划分区域对曲面积分，所得到的相对强度值，下面给出在 8×8 变换情况下一个典型的加权系数矩阵。

$$\begin{bmatrix}
0.4942 & 1.0000 & 0.7023 & 0.3814 & 0.1856 & 0.0849 & 0.0374 & 0.0160 \\
1.0000 & 0.4549 & 0.3085 & 0.1706 & 0.0845 & 0.0392 & 0.0174 & 0.0075 \\
0.7023 & 0.3085 & 0.2139 & 0.1244 & 0.0645 & 0.0311 & 0.0142 & 0.0063 \\
0.3814 & 0.1706 & 0.1244 & 0.0771 & 0.0425 & 0.0215 & 0.0103 & 0.0047 \\
0.1856 & 0.0845 & 0.0645 & 0.0425 & 0.0246 & 0.0133 & 0.0067 & 0.0032 \\
0.0849 & 0.0392 & 0.0311 & 0.0215 & 0.0133 & 0.0075 & 0.0040 & 0.0020 \\
0.0374 & 0.0174 & 0.0142 & 0.0103 & 0.0067 & 0.0040 & 0.0022 & 0.0011 \\
0.0160 & 0.0075 & 0.0063 & 0.0047 & 0.0032 & 0.0020 & 0.0011 & 0.0006
\end{bmatrix}$$

如果在变长编码情况，考虑 HVS 特性，应对不同的变换系数 $y(k,l)$，采用不同的量化步长进行均匀量化，即对每个变换系数仍采用均匀量化加熵编码，但每个均匀量化器的步长是随频率变化的，量化步长Δ不再取常数，而是(k,l)的函数，即$\Delta(k,l)$。量化步长$\Delta(k,l)$与加权系数 $w(k,l)$ 是一种反比关系，在 JPEG 编码标准中，对于 8×8 块亮度分量 DCT 变换系数推荐的一种量化步长矩阵为：

$$\boldsymbol{Q}=[\Delta(k,e)]_{8\times 8}=\begin{bmatrix}16 & 11 & 10 & 16 & 24 & 40 & 51 & 61\\ 12 & 12 & 14 & 19 & 26 & 58 & 60 & 55\\ 14 & 13 & 16 & 24 & 40 & 57 & 69 & 56\\ 14 & 17 & 22 & 29 & 51 & 87 & 80 & 62\\ 18 & 22 & 37 & 56 & 68 & 109 & 103 & 77\\ 24 & 35 & 55 & 64 & 81 & 104 & 113 & 92\\ 49 & 64 & 78 & 87 & 103 & 121 & 120 & 101\\ 72 & 92 & 95 & 98 & 112 & 100 & 103 & 99\end{bmatrix} \tag{2.50}$$

2.6.5　量化系数的扫描和表示方法

接下来，进一步讨论均匀量化加熵编码的一些具体实现技术，为了下面讨论清楚，首先给出一个实例，从 lena 图像中，取一个 8×8 块，进行变换、量化、反量化和反变换等过程，因为熵编码并不引入误差，反变换得到的就是实际解码图像。

一个平坦的图像块为：

80	69	71	75	79	84	89	91
82	69	70	73	76	83	90	95
79	77	74	76	74	85	89	95
80	71	73	76	79	86	91	93
84	74	77	77	82	88	91	93
84	78	76	80	84	88	92	95
85	76	78	80	85	93	94	95
85	74	79	81	85	86	94	94

以下是它的 DCT 变换系数，可以看到能量集中在少数低频系数。

660.125 0	−47.049 6	25.998 0	10.399 3	7.875 0	8.486 6	5.602 5	1.317 6
−17.326 7	−2.674 9	5.223 6	−1.323 4	0.522 2	0.291 4	0.280 0	−2.281
0.028 0	−0.646 3	−0.954 5	0.962 0	2.473 0	1.978 3	−0.316	2.174 1
2.300 3	0.454 2	−2.240 3	3.555 9	1.290 7	−1.002 4	0.158 0	0.974 7
−2.375 0	0.103 8	−3.222 0	0.965 3	1.375 0	2.225 8	0.387 5	3.523 6
0.929 4	−1.328 2	−2.425 6	0.982 8	−1.931 7	−0.697 2	0.125 3	−1.856
0.394 3	2.664 0	−0.566 9	−3.416 8	−0.889 1	−1.618 2	−2.545	−1.732
2.166 6	1.723 8	−0.333 5	−0.480 8	−2.625 3	−0.969 9	1.485 4	−1.183

用 JPEG 的亮度量化矩阵（2.50）对每个系数进行均匀量化，量化器输出序号为：

41	−4	3	1	0	0	0	0
−1	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0

0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0

反量化后，进行 DCT 反变换，得到的解码图像为：

80	75	71	72	78	85	89	90
80	75	71	72	78	85	89	90
80	76	72	73	79	86	90	91
81	77	72	74	80	87	91	92
82	77	73	74	81	87	91	93
83	78	74	75	81	88	92	93
83	79	75	76	82	89	93	94
84	79	75	76	82	89	93	94

经计算得到均方失真为 7.2031。

由这个例子看到，由于变换后能量主要集中在很少几个变换系数上，所以产生大量 0 系数，正如我们在前面讨论的，量化器的输出由非零值幅度和非零值位置两部分组成，一种简单的编码方法是分别产生位置符号串和幅度符号串，并对两个符号串分别进行熵编码（Huffman 或算术编码，）对这个例子，位置串为 [0,0],[0,1],[0,2],[0,3],[1,0]，幅值串为 41，−4，3，1，−1。

对 DCT 变换编码，处理量化器输出符号的更为有效的方法是扫描和零行程编码，扫描是按一种前进次序将量化后二维变换系数形成一个串，然后对这个串进行零行程编码，最常用的扫描方式是 Zig-Zag 扫描，8×8 DCT 变换量化系数的 Zig-Zag 扫描次序示于图 2.22。

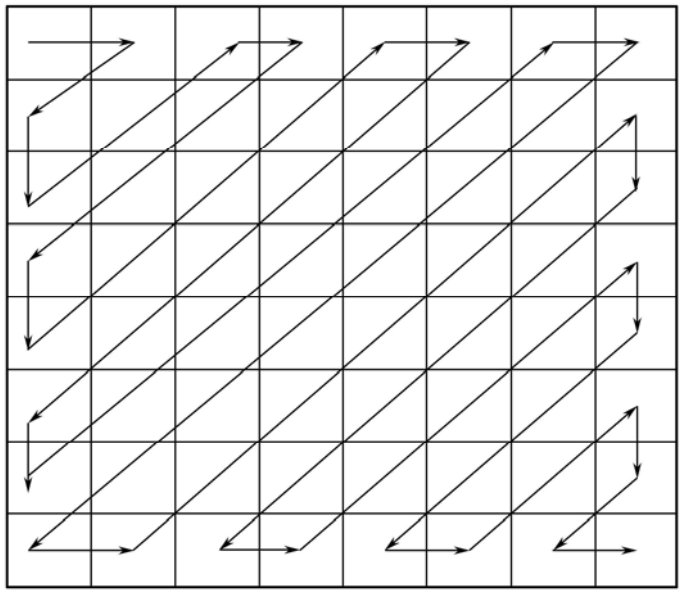


图 2.22　DCT 变换系数的 Zig-Zag 扫描次序

例子所示的量化后系数的 Zig-Zag 扫描为 41，−4，−1，0，0，3，1，EOB，EOB 是块结束符，它指示在扫描过程中，已经遇到最后一个非零值，对扫描序列，形成[*L*，*V*]结构的[行

程，符号]为[0,41], [0, −4], [0, −1], [2,3], [0,1], [0,EOB]。零行程序列中，行程 L 和幅度 V 分别采用 Huffman 或算术编码进行熵编码。

注意在实际编码器设计时，通过大量的样本图像统计或通过模型数据，获得行程和幅度的概率表，然后形成 Huffman 码表，作为编码器的固定参数表，或用自适应算术编码进行熵编码。

2.6.6　一个编码实例

下面用一个简单的 DCT 变换编码器说明变换编码的步骤，图像被分割成 8×8 小块，每一块单独进行 DCT 变换，每个变换系数采用同一个量化步长Δ进行均匀量化，对一个变换系数 $\hat{y}$（k,l），量化器输出序号为：

$$Q(k,l) = \text{round}(y(k,l)/\Delta)$$

在解码器，变换系数重构为：

$$\hat{y}(k,l) = Q(k,l)\cdot\Delta$$

所有块的变换系数分成两组，DC 系数和 AC 系数，分别采用两个独立的自适应算术编码器，对于 256×256 的 Lena 测试图像，几个典型码率下的解码图像如图 2.23 所示。

图 2.23　解码图像

2.7　块匹配运动估计与补偿

在视频图像压缩编码中，图像是由连续的帧形成的图像序列，由于景物变化速度的限制，相邻帧间存在很高的相关性，即存在很高的时域冗余。怎样利用这种冗余，达到更高的压缩效率，是序列图像编码的主要问题。

运动补偿预测的目的就是消除时域冗余，由运动补偿技术结合变换编码构成序列图像编码的主要方法。本节首先较详细地讨论运动补偿技术，重点讨论实用的块匹配运动估计与补偿方法，下一节再结合变换编码技术构成序列图像编码器。

序列图像中，相邻帧间的主要变化是由于构成景物的诸物体的运动引起的，检测物体的运动参数，并通过这些运动参数由前一帧预测当前帧，这就是运动补偿预测 MCP（Motion Compensated Prediction），或简称运动补偿（MC），其中主要的任务是检测物体的运动参数，这称为运动估计 ME（Motion Estimation）。

物体的运动由多种元素构成，包括平移、旋转，扭曲等，完整描述物体运动的模型是复杂的，在目前通用图像压缩标准及大多数实用的图像压缩算法中，均使用简化的运动模型，即假设运动是由平移构成，这样只用 X 和 Y 方向（或水平与垂直方向）的两个平移参数 $\mathrm{d}x,\mathrm{d}y$ 表征运动参数。由 $\mathrm{d}x,\mathrm{d}y$ 构成一个运动矢量 $\boldsymbol{D}$：

$$\boldsymbol{D}=(\mathrm{d}x,\mathrm{d}y)^{\mathrm{T}}$$

当前帧（i,j）坐标点的像素值由前一帧预测为：

$$\mathrm{MCP}(i,j,k)=\hat{S}(i,j,k)=S(i+\mathrm{d}x,j+\mathrm{d}y,k-1) \tag{2.51}$$

这里 k 表示当前帧，k–1 表示前一帧，$S(i,j,k)$ 表示原始像素值，$\hat{S}(i,j,k)$ 表示预测的像素值。

当前帧原始图像与预测图像之间可能会有误差，由于这个误差是由平移运动补偿预测引起的，故称为帧间位移误差 DFD（Displacement Frame Difference），表示如下：

$$\begin{aligned}\mathrm{DFD}(i,j,k)&=S(i,j,k)-\hat{S}(i,j,k)\\&=S(i,j,k)-S(i+\mathrm{d}x,j+\mathrm{d}y,k-1)\end{aligned} \tag{2.52}$$

有了这些结果，我们将对图像帧内像素的编码变成对 DFD 场和运动矢量 $\boldsymbol{D}$ 的编码，由于相邻帧的高相关性，运动补偿是有效的，DFD 场能量非常低，可以用很少的码字表示；运动矢量是稀疏的，同样用较少码字表示，这样就可以获得有效的码率压缩。

这里，平移运动假设的有效性是有条件的，如果一个运动物体较大时，将它分成一些小块，对于每个小块的平移与旋转运动可以用一个平移运动矢量来逼近，如果进行分块运动估计，则每个块的运动矢量可以逼近平移和旋转运动，所以块匹配运动估计是目前应用最广泛的，本章主要以块匹配方法为主，介绍运动估计与补偿的基本算法，然后，结合运动补偿技术和变换编码技术，构造有效的序列图像编码算法。

对于当前的第 k 帧图像，将它划分成等大小的块，每个块的尺寸为 $N\times M$，即每个块包含 N 行，每行 M 个像素，每个块起始点在整帧图像中的行、列标号为（I,J），整个块用符号 B（I, J, k）表示，假设相邻帧间的各方向的最大运动矢量为 $d_{\max}$，这样就在第 k–1 帧构成一个搜索窗，这个窗的中心点坐标与 B（I, J, k）在 k 帧的中心点坐标重合，窗大小为

$(2d_{\max}+N)\times(2d_{\max}+M)$。块 B（I, J, k）的运动补偿预测块必定是窗内以某一位置为起点的一个块，设运动矢量为 $(\mathrm{d}x,\mathrm{d}y)^{\mathrm{T}}$，则 B（I, J, k）的运动补偿块为 $B(I+\mathrm{d}x,J+\mathrm{d}y,k-1)$，运动估计的目的是求这一矢量 $(\mathrm{d}x,\mathrm{d}y)^{\mathrm{T}}$，块匹配的示意图如图 2.24 所示，图的下侧代表当前帧，待处理的块用粗线表示，图的上侧代表前一帧，虚线框住的是搜索窗，最大搜索距离 $d_{\max}=W$，在搜索窗中的最好匹配块也用粗线表示。

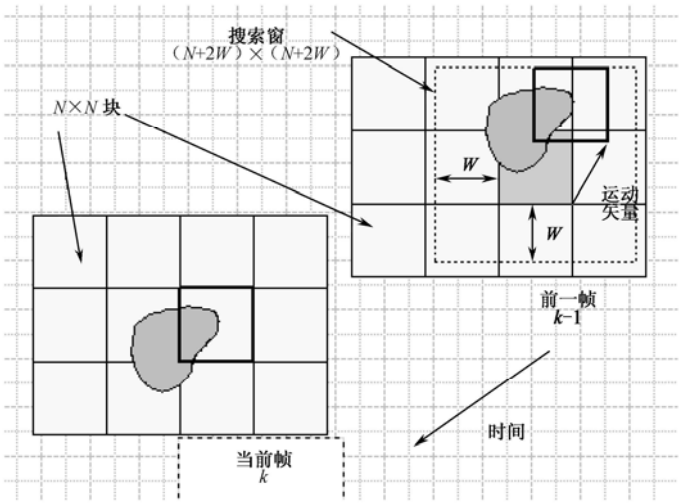


图 2.24　块匹配示意图

很明显，在搜索窗中所能构成的所有块的集合为：

$$\{B(I+i,J+j,k-1)-d_{\max}\leqslant i,j\leqslant d_{\max}\}$$

在所有这些块中，与 $B(I,J,k)$ 最接近的，或者说按某一准则最匹配的块，假设为 $B(I+i_0,J+j_0,k-1)$，就是我们要寻求的 $B(I,J,k)$ 的运动补偿块，$(i_0,j_0)^{\mathrm{T}}$ 就是运动矢量。必须要定义一个匹配准则，用于描述两个块的接近性，文献中常用的匹配准则是均方误差准则（MSE）和平均绝对差值准则（MAD）。块 $B(I,J,k)$ 和 $B(I+i,J+j,k-1)$ 的 MSE 和 MAD 度量定义分别为：

$$\mathrm{MSE}(i,j)=\frac{1}{MN}\sum_{h=0}^{N-1}\sum_{m=0}^{M-1}(S(I+n,J+m,k)-S(I+i+n,J+j+m,k-1))^2 \tag{2.53}$$

$$\mathrm{MAD}(i,j)=\frac{1}{MN}\sum_{n=0}^{N-1}\sum_{m=0}^{M-1}\left|\begin{matrix}S(I+n,J+m,k)\\-S(I+i+n,J+j+m,k-1)\end{matrix}\right| \tag{2.54}$$

由这两个准则获得的运动估计与预测效果相当接近，由于 MAD 准则不需要乘法运算，运算量小得多，故采用得更多一些。这两个准则都属于最小准则，取值越小越匹配，还有一些其他的最小准则也有人采用。

另外一类准则是最大准则，例如，两块之间的相关系数准则。在本书中，运动估计算法只使用最小准则。为了通用，在讨论运动估计算法时，不指定一种具体准则，而使用一个通用准则符号 Crit。定义如下：

$$\mathrm{Crit}(i,j)=\begin{cases}\mathrm{MSE}(i,j) & \text{若采用MSE准则}\\ \mathrm{MAD}(i,j) & \text{若采用MAD准则}\\ \text{或其他准则}\end{cases}$$

最简单也是最准确的运动估计算法是全搜索算法，对所有 $-d_{\max}\leqslant i,j\leqslant d_{\max}$ 计算 Crit（i,j）得到一个集合：

$$\{\mathrm{Crit}(i,j),\quad -d_{\max}\leqslant i,j\leqslant d_{\max}\}$$

该集合中的最小值发生在（i,j）取值为（i_0,j_0）时，即：

$$\mathrm{Crit}(i_0,j_0)=\min_{-d_{\max}\leqslant i,j\leqslant d_{\max}}\{\mathrm{Crit}(i,j)\} \tag{2.55}$$

那么 $(i_0,j_0)^{\mathrm{T}}$ 就是所求运动矢量，有：

$$\boldsymbol{D}=(\mathrm{d}x,\mathrm{d}y)^{\mathrm{T}}=(i_0,j_0)^{\mathrm{T}}$$

式（2.55）表示的全搜索运动估计算法，对每一个块需要计算匹配准则 $(2d_{\max}+1)^2$ 次，当 $d_{\max}$ 较大时，运算量非常巨大。

一旦运动矢量 $\boldsymbol{D}$ 得到后，块 $B(I,J,k)$中每个像素的运动补偿预测值为：

$$\mathrm{MCP}(I+i,J+j,k)=S(I+\mathrm{d}x+i,J+\mathrm{d}y+j,k-1) \tag{2.56}$$

位移帧间差值为：

$$\begin{aligned}\mathrm{DFD}(I+i,J+j,k)&=S(I+i,J+j,k)-\mathrm{MCP}(I+i,J+j,k)\\&=S(I+i,J+j,k)-S(I+\mathrm{d}x+i,J+\mathrm{d}y+j,k)\end{aligned} \tag{2.57}$$

在式（2.56），式（2.57）中，$0\leqslant i\leqslant N-1,0\leqslant j\leqslant M-1$。

有了这些结果以后，对块 $B(I,J,k)$的编码，转化为对块内位移帧间差值场 $\{\mathrm{DFD}(i,j),I\leqslant i<I+N,J\leqslant j<J+M\}$和运动矢量 $\boldsymbol{D}$ 的编码，对于运动估计有效的块，DFD 能量可忽略，这样的块只需要编码一个运动矢量，只有对 DFD 能量不可忽略的块，才对其 DFD 进行编码。

鉴于全搜索算法的运算复杂性太高，以下介绍几种快速算法。

2.7.1　运动矢量的快速搜索算法

对 $\mathrm{Crit}(i,j)$ 不作任何假设的全搜索算法，必然能够找到 $\mathrm{Crit}(i,j)$ 的全局最小点，如果对 $\mathrm{Crit}(i,j)$ 作一些假设，可以得到更有效的搜索算法，一种假设是：$\mathrm{Crit}(i,j)$ 在其定义域上只有一个极小点 (i_0j_0)，随着偏离 (i_0j_0) 距离的增加，$\mathrm{Crit}(i,j)$ 单调增，在这个假设下，很自然导出如下几种搜索算法。

1）算法 1：二维对数搜索算法

图 2.25 构成一个（i,j）栅格，每个栅格的交点对应一个 $\mathrm{Crit}(i,j)$ 的值，二维对数搜索算法的思路是：以（0,0）点为初始搜索中心，逐步朝着 $\mathrm{Crit}(i,j)$ 的极小点位置逼近。在每一步，有 5 个（i,j）位置被检查，找出使 $\mathrm{Crit}(i,j)$ 最小的点，如果前一步的最小点处在 5 个搜索位置的中心，或处在搜索区域的边界，则搜索点的距离降低，在图 2.25 的例子中，$d_{\max}=7$，通过 5 步，得到运动矢量为（−6，3）。这是运动矢量接近边界的情况，也是需要搜索次数较多接近最坏的情况。

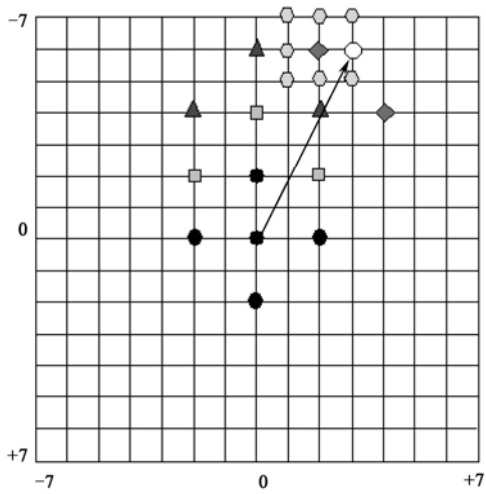


图 2.25　二维对数搜索算法

下面先定义的两个集合，然后给出算法的正规描述：

$$N(m)=\{(i,j),-m\leqslant i,j\leqslant m\}$$

$$M(m)=\{(0,0),(m,0),(0,m),(-m,0),(0,-m)\}$$

算法描述如下。

（1）算法初始化：

$$n'=\lfloor \mathrm{lb}\, d_{\max}\rfloor$$

$$n=\max\{2,2^{n'-1}\}$$

$$i_0=j_0=0$$

（2） $M'(n)\leftarrow M(n)$

（3）找到 $(i,j)\in M'(n)$，使 $\mathrm{Crit}(i+i_0,j+j_0)$ 最小。如果 *i=0,j=0*，转入（5），否则转到（4）。

（4） $i_0\leftarrow i_0+1, j_0\leftarrow j_0+1$， $M'(n)\leftarrow M'(n)-(-i,-j)$，转到（3）。

（5） $n\leftarrow n/2$，如果 *n*=0，转（6），否则转到（2）。

（6）找到一个（*i,j*）$\in N$（1）使 $\mathrm{Crit}(i+i_0,j+j_0)$ 最小，$i_0\leftarrow i_0+1, j_0\leftarrow j_0+1$，这里（$i_0$,$j_0$）即为所求运动矢量。

2）算法 2：三步搜索算法

图 2.26 表示三步搜索的一个例子。三步搜索和对数搜索思路很接近，只是每次搜索要对中心点及其周围的 8 个点进行。起始时，两个搜索点距离为（$d_{\max}$+1）/2，每步搜索距离减半，在例子中，对于 $d_{\max}$=7 的情况只需三步，故称三步法，其实随着 $d_{\max}$ 增加，步数会增多。以图 2.26 为例，第一步，搜索点距离为 *d*=4，设 Crit(0,4)最小；第二步时，令 *d*=2，周围 8 个点与中心点比较，设 Crit(−2,4)最小；第三步时，令 *d*=1，比较结果 Crit(−3,5)最小，故 $\boldsymbol{D}=(-3,5)^{\mathrm{T}}$。如果 $d_{\max}$=15，则需要四步，搜索点距离分别为{8，4，2，1}。

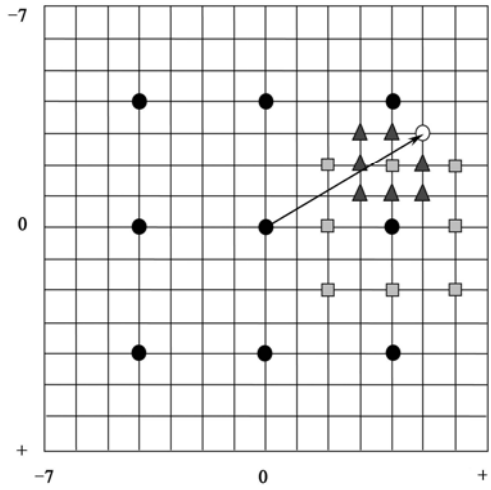


图 2.26　三步搜索算法

三步法是一个应用很广泛的算法，它的搜索次数少，大多数情况可以获得可接受的搜索性能，为了更好地搜索速度或性能，不少研究者提出一些对三步法的改进，较常用的是新三步法和四步法。

3）算法 3：新三步法

新三步法的特点是在第一次搜索时增加一个内环，搜索可能的最小点，第一次搜索的布局点见图 2.27。有几种常见情况发生，情况 1，第一次搜索的最小点在外环，这种情况下，内环没有起作用；情况 2，第一次搜索的最小点在内环的中点，这种情况令运动矢量为零；情况 3，第一次搜索的最小点在内环上，这种情况以最小点为中点再进行一次距离为 1 的全搜索。情况 1 和情况 3 分别示于图 2.28 和图 2.29。三种情况搜索次数分别为 33、17 和 22，可以看到，在运动比较小的序列，新三步法有更低的运算量，对于激烈运动序列，它反而比三步法慢，类似的改进研究还有四步法等，不再赘述。

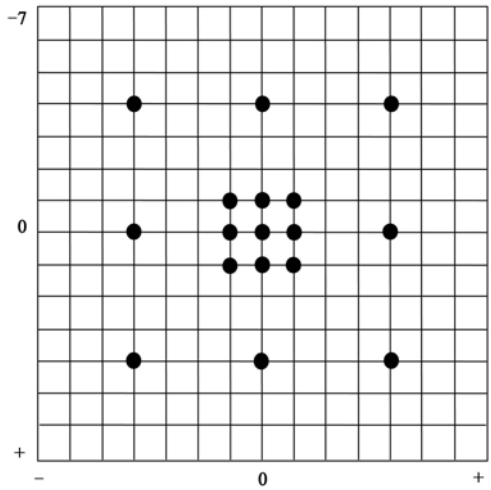


图 2.27　新三步法的内环模式

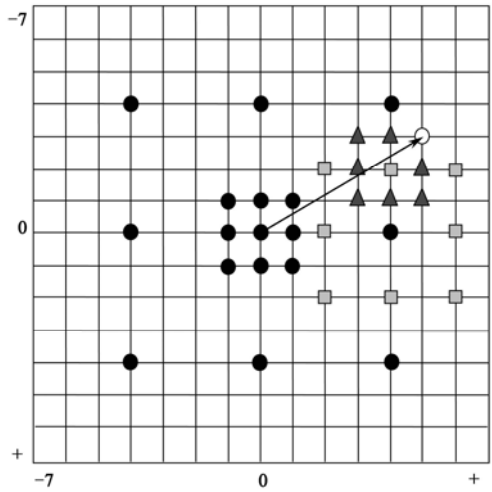


图 2.28　新三步法最小点在外环的情况

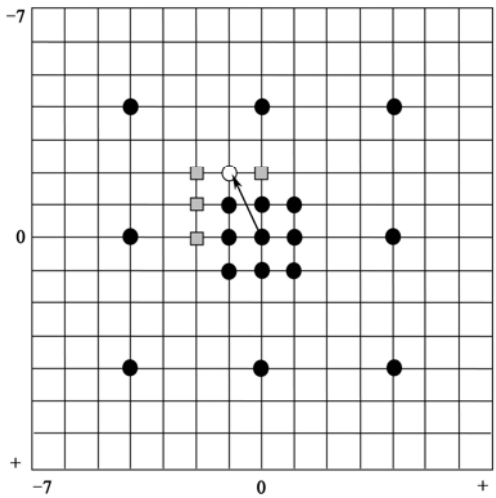


图 2.29　新三步法第一次搜索最小点在内环的情况

4）算法 4：钻石形搜索

根据运动可能是各向同性的特点，一种非矩形搜索模式被提出来，它就是钻石形搜索，它的一个搜索图形的实例见图 2.30。一般钻石搜索可以获得比三步法等更好的性能，但运算复杂性也要高一些。

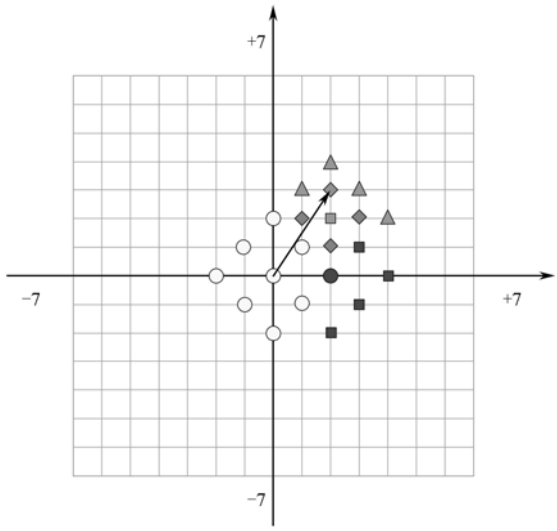


图 2.30　钻石形搜索实例

在满足两个假设（极点唯一和单调性）的条件下，快速搜索算法与全搜索算法收敛在同一个点上，但对实际图像 Crit(*i*, *j*)的立体图示表明，它并不满足这两个假设，实际中，快速算法经常会收敛于一个局部极小点上而不是全局最小点。因此，快速算法会有一定的性能损失。

表 2.12 是对几个 $d_{\max}$ 取值时，全搜索法与快速算法的搜索次数比较，表上列出的均是快速算法在最坏情况的搜索次数。

表 2.12　快速搜索法与全搜索算法搜索次数比较

算法 \ $d_{\max}$	6	15	31
三步法	25	33	41
对数法	21	30	37
共轭方向	15	35	65
全搜索	169	108 9	396 9

2.7.2　变块大小的分层运动估计

在分块运动估计的算法中，块大小参数的选择是值得考虑的问题，块选择得过大，每个块中可能包含几个运动物体，降低运动预测的有效性；块太小时，运动预测容易受其他因素的控制，如像素点的噪声，或局部相似区域等，尽管可能使 Crit(*i*, *j*)很小，但却不一定能够得到真实的运动矢量，从而使运动矢量场分布极不规则，增大运动矢量场的熵值，使对运动矢量的编码效率不高。为了解决这一问题，一些学者提出了变块大小和多分层的运动估计方法。

M．Bierling 首先提出了一种分层块匹配的运动估计方案。在他的算法中，运动矢量场的估计是通过在多层中依次完成块匹配而获得的。在高层中，匹配使用较大的区域以估计图像

的总体运动趋势，为了降低运算量，对原图像进行低通滤波和亚采样；在低层中，使用较小的块，逼近局部的运动。Bierling 提出的这种分层估计的思想，被进一步完善，形成多种分层运动估计方法，以下介绍两种典型的实现方案。

1）多格点块匹配算法

图 2.31 是多格点块匹配算法的示意图，在最顶层，按 $N\times M$ 对图像进行固定大小分块，对每一块进行运动估计，搜索范围为 d_L（L 为最大分层数）；得到顶层运动矢量 $mv_L=(i_L,j_L)^{\mathrm{T}}$，和最小准则值 $\mathrm{Crit}(i_L,j_L)$。如果 $\mathrm{Crit}(i_L,j_L)$ 小于一个预置的门限值，则该块停止；如果 $\mathrm{Crit}(i_L,j_L)$ 大于预置门限值，这一块将分成 4 个 $\frac{N}{2}\times\frac{M}{2}$ 大小的子块进入下一层的运动估计。在最高层运动估计完成后，有些块已满足要求的运动补偿预测精度，不再进入下一层，而不满足精度的块分成 4 个子块进入下一层估计。子块相应的上一层的块称为父块。在次高层（$L-1$ 层），每一子块以父块的运动矢量为初始值，在 d_{L-1} 的搜索范围内进行匹配，寻找最匹配点；这个过程一直进行下去，直到底层（第一层）。这种分层方法的最大搜索范围为：

$$d_{\max}=\sum_{i=1}^{L}d_i \tag{2.58}$$

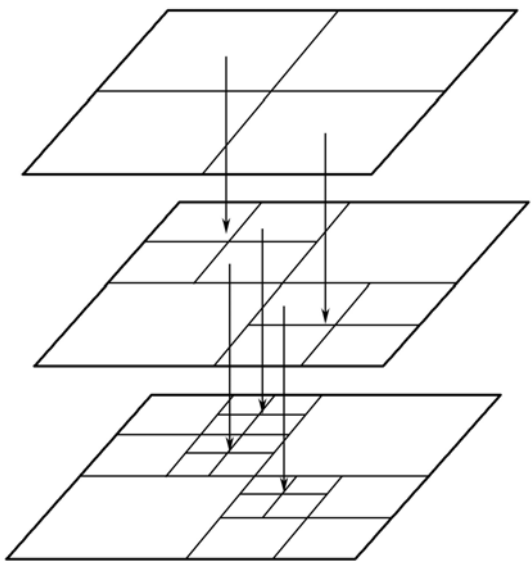


图 2.31　多格点块匹配算法示意图

最坏情况（上层的每一块均需要继续分解到下一层）情况下，搜索次数为 C_s（指在每一层均采用全搜索算法）：

$$C_s=\sum_{i=1}^{L}(2d_i+1)^2 \tag{2.59}$$

例如，$d_{\max}$=15,令 $d_1=4,d_2=5,d_3=6$ 则 $C_s=\sum_{i=1}^{3}(2d_i+1)^2=371$，比直接搜索的 961 次节省 60%的运算量。

2）多分层塔结构运动估计

对当前帧图像和前一帧图像分别构造多层塔，每向上一层，图像横向和纵向尺寸减半，一个三层图像塔的结构如图 2.32 所示。

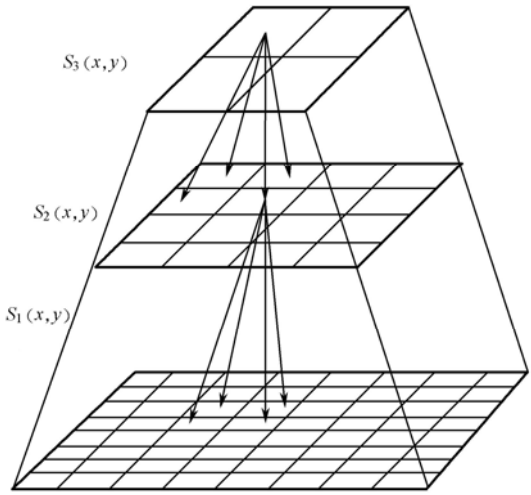


图 2.32　三层图像塔的结构

塔的结构按递推进行，由 $S_1(x,y)$ 构造 $S_2(x,y)$，由 $S_2(x,y)$ 构造 $S_3(x,y)$，依次递推，最简单的一种均值塔构造算法如下：

$$S_{l+1}(x,y)=\frac{1}{4}\sum_{i=0}^{1}\sum_{j=0}^{1}S_l(2x+i,2y+j) \tag{2.60}$$

在这种塔式构造中，每一层的分块大小都是相等的，例如，每层均按 16×16 式 8×8 分块，为了一般性，设为 $N\times M$，每一层的块数目 B_l 满足 $B_{l+1}=\frac{1}{4}B_l$。每一个上层块作为父块对应下层 4 个子块，这些关系如图 2.32 所示。

塔式运动估计从顶层开始，搜索范围为 d_L，对每一个块进行运动估计，对一个块得到运动矢量为 $\mathbf{mv}_L=(i_L,j_L)^{\mathrm{T}}$，其最小准则值为 $\mathrm{Crit}(i_L,j_L)<T_L$。如果 $\mathrm{Crit}(i_L,j_L)<T_L$，则这一块运动估计终止在这一层，这里 T_L 为预定义的门限值，对于已确定为停止的块，将运动矢量 $(2i_L,2j_L)^{\mathrm{T}}$ 赋予 4 个子块，$(4i_L,4j_L)^{\mathrm{T}}$ 赋予 16 个更下层的子块，直到将运动矢量 $(2^{L-1}i_L,2^{L-1}j_L)^{\mathrm{T}}$ 赋予 4^{L-1} 个最底层块。除顶层块外，向下层的每一层中，对于父块不属于停止块的那些块，均以上层父块运动矢量的 2 倍作为子块运动矢量的初值，以 d_l 为范围作运动估计，判断停止块和继续块，其他过程同顶层，这个递推过程直到最底层。

这个过程同样得到多分辨率的运动矢量场，在 L 层的一个运动矢量乘 2^{L-1} 倍赋给最底层的 4^{L-1} 个块，相当于 $2^{L-1}N\times 2^{L-1}M$ 的像素区域。图 2.33 为最终在底层得到一个运动矢量场的块结构的例子。

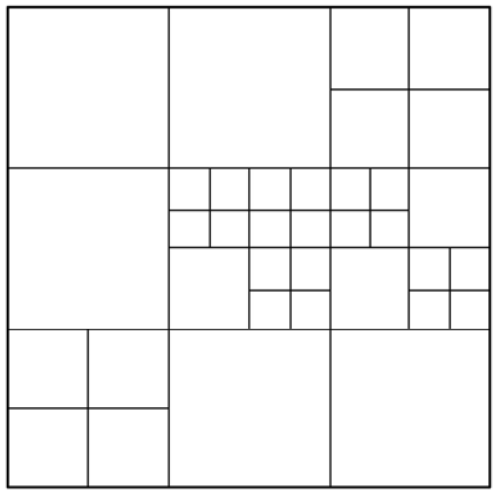


图 2.33　塔结构运动估计最终矢量场的块结构

塔结构运动估计算法的总运动搜索范围 $d_{\max}$ 为：

$$d_{\max}=\sum_{i=1}^{L}2^{i-1}d_i \tag{2.61}$$

如果每一层均采用全搜索算法，则总搜索次数为 C_s：

$$C_s=\sum_{i=1}^{L}(2d_i+1)^2 \tag{2.62}$$

例如，$d_{\max}$=16，取 $d_1=2,d_2=3,d_3=2$，$C_s=\sum_{j=1}^{3}(2d_i+1)^2=99$，与直接单层全搜索的次数 1 089 比较，节省 91%的搜索次数。

下面对分层运动估计的几个公共问题进一步说明。

1）运动初始值选择问题

除顶层以外的其他层运动估计，均要从上层继承一个运动初始值，最简单的方法是直接从父块获得运动初值，以上两种算法的描述过程都是这样的，但在一个父块中包含两个运动物体时，这样并不是最好的，图 2.34 表示了这种情况。

在图 2.34 中，中间的块需要分解成 4 个子块作进一步运动估计，如果 4 个子块直接继承父块矢量，则如图 2.34（b）的情况，其实右上角的初始矢量是错误的，一个解决办法是，以父块运动矢量及该子块在父块中与之相邻的 3 个上层块的运动矢量作为备选初始矢量，计算 Crit，使 Crit 最小的备选初始矢量最终作为该子块要继承的初始矢量，图 2.34（c）就是这样得到的 4 个子块的初始矢量分布，它们都正确地代表每一子块的运动趋势。

2）准则选择问题与块停止判决

在以上两个分层算法的描述过程中，一个块的运动估计是否在当前层停止，使用简单的判决准则：

$$\mathrm{Crit}(i_L,j_L)<T \tag{2.63}$$

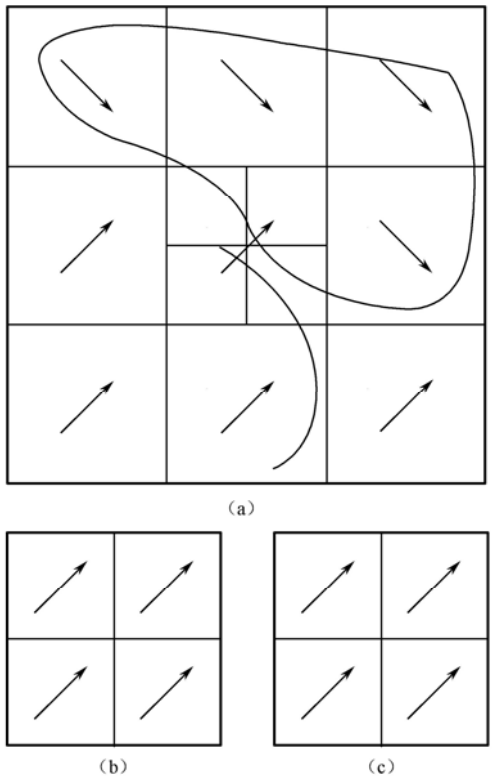


图 2.34　运动初值选择

如果满足上述准则，则停止，否则分解成 4 个子块，继续进行运动估计。这个准则只是简单易用，却不是最优的准则，最优的准则应该是判断继续分解是否最终获得编码码字的降低，等价的是相应熵的降低，一个更准确的做法是，除非最底层，否则总要将当前层的每一块分解成子块作进一步运动估计，然后将上、下两层得到的运动矢量都映射到最底层中，并代入式（2.52）作运动补偿预测，获得相应的 DFD 场，分别记为 DFD_L 和 DFD_{L-1}，对 DFD_L 和 DFD_{L-1} 进行概率统计，得到其取值的概率分布表，计算它们各自的熵 H_{DL} 和 H_{DL-1}。运动矢量也进行同样的概率统计 H_{VL} 和 H_{VL-1}，如果满足下式：

$$4H_{VL-1}+n\cdot H_{DL-1}>H_{VL}+n\cdot H_{DL} \tag{2.64}$$

则说明继续分解获得的运动估计增加了编码码字，因此，继续分解是不必要的，运动估计将在该层停止，这里 n 表示底层相应块中像素数目。

以上的准则是局部最优的，但运算量太大，实际中经常选择一些折中的准则，例如：

$$\mathrm{Crit}_l<\frac{1}{4}\sum_{i=1}^{4}\mathrm{Crit}_{l-1,i}-\delta \tag{2.65}$$

则判决停止，这里δ 是对不同类型的序列（如会议电视，广播电视，可视电话等），通过反复实验确定的一个参数。这种方法在 H.263 的一个实现模型中，给出了一个实际的参数。

3）四叉树表示分层结构

分层运动估计获得的运动矢量，不管停止在哪一块，均可以映射到最底层块，从而对每

一个底层块传送一个运动矢量，这是最简单的处理方法。对分层过程的分解结构用树形表示，只传送树结构和停止块的运动矢量，可以获得更高的效率。图 2.35 表示一个块分解过程及其相应的四叉树表示，对每个停止的块，节点标为 0，继续分解的块节点标为 1，只对相应的零节点传送运动矢量，注意，整个图像被赋予为根节点。

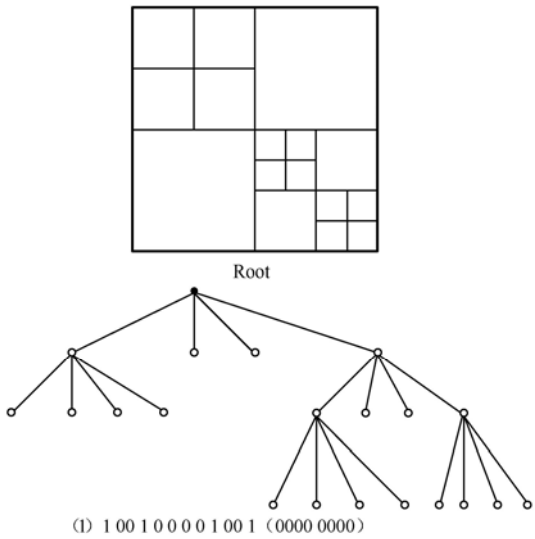


图 2.35　块分解过程及分层结构的四叉树表示

图 2.35 中，表示四叉树的码是按层形成的，第 1 个码位表示根节点，接下来 4 个码位表示第 1 层的 4 个块，由于第 2、3 两块运动矢量估计在该层停止，因此为零节点，1，3 两块将在下一层中分解成 4 个子块进行运动估计。在四叉树表示中，由于根节点总为 1，而最低层的节点总为零节点，故可以省略（图中括号内部分）。图 2.35 所示结构中，只需传送 16 个运动矢量和 100100001001 这样一个树结构码，如果直接将运动矢量映射到最低层，则需要传送 64 个运动矢量。

2.7.3　分数像素运动估计

数字图像是对模拟图像按横向和纵向 X、Y 为距离采样获得，每个像素的位置位于 (nX, mY) 点上。在数字图像表示中，为了方便，将这个像素的坐标简写为 (n, m)，这里 n,m 为整数。很显然，前面介绍的运动估计方法得到的运动矢量均为 X，Y 的整数倍，我们称为整像素精度运动估计，或简称为整数运动估计。对于一个获得的运动矢量 $\mathbf{mv} = (i_0, j_0)^{\mathrm{T}}$，对应到模拟空间为矢量 $(i_0 X, j_0 Y)^{\mathrm{T}}$。由于实际图像中物体的运动并不一定是 X，Y 的整数倍，因此，整数运动估计是不准确的。

为了获得精确的运动估计，第一步是对前一帧图像进行插值，得到逼近的模拟图像，一般的插值公式为：

$$\tilde{S}(x,y,k) = \sum_{(x_s,y_s)\in\Pi} \tilde{f}(x-x_s, y-y_s)\cdot S(x_s,y_s,k) \qquad (2.66)$$

$$\forall (x,y) \in R^2$$

式中，$S(x_s, y_s, k)$ 为数字图像在整数坐标 (x_s, y_s) 点的值，k 是帧序号，Π 表示一幅数字图像所有坐标点的集合，$\tilde{f}(x, y)$ 是插值的核函数，$\tilde{s}(x, y, k)$ 是插值得到的逼近的模拟图像在任意实数坐标点 (x, y) 的值。在实际操作时，要取一个具体的插值核函数，以构成一个具体的插值公式，最著名的插值核函数是 sinc 函数。

$$\tilde{f}(x, y) = \frac{\sin(\pi x / X)}{(\pi x / X)} \cdot \frac{\sin(\pi y / Y)}{(\pi y / Y)} \tag{2.67}$$

插值公式写成：

$$\tilde{S}(x, y, k) = \sum_{(x_s, y_s) \in \Pi} \frac{\sin \frac{\pi(x - x_s)}{X} \cdot \sin \frac{\pi(y - y_s)}{Y}}{\frac{\pi(x - x_s)}{X} \cdot \frac{\pi(y - y_s)}{Y}} \cdot S(x_s, y_s, k) \tag{2.68}$$

如果数字图像在获取时，满足采样定理，即 X，Y 分别小于图像横向和纵向最高频率 2 倍的倒数，由式（2.68）获得的模拟图像是对原始模拟图像的准确重构（在不计入量化误差的情况下）。但是这个重构公式的计算是非常复杂的，不适合于实时的实现，既使取其前几项的近似逼近，也仍嫌复杂，实际中常用的是双线性插值，其核函数为：

$$\tilde{f}(x, y) = \max\left\{0, 1 - \left|\frac{x}{X}\right|\right\} \cdot \max\left\{0, 1 - \frac{y}{Y}\right\} \tag{2.69}$$

对于 $(n-1)X \leqslant x \leqslant nX (m-1)Y \leqslant y \leqslant mY$，插值公式简化为：

$$\begin{aligned} \tilde{S}(x, y, k) = {} & (n - \frac{x}{X})(m - \frac{y}{Y}) \cdot S(n-1, m-1, k) + \\ & (n - \frac{x}{X})(1 - m + \frac{y}{Y}) \cdot S(n-1, m, k) + \\ & (1 - n + \frac{x}{X})(m - \frac{y}{Y}) \cdot S(n, m-1, k) + \\ & (1 - n + \frac{x}{X})(1 - m + \frac{y}{Y}) \cdot S(n, m, k) \end{aligned} \tag{2.70}$$

在实际图像编码器中，并不需要无限精度的运动矢量，图像压缩编码的目的，是在保持同样图像质量的情况下，尽可能地降低码率。随着运动矢量精度提高，运动预测补偿更有效，DFD 场能量进一步降低，但运动矢量编码需要更多码字。一旦 DFD 场降低的码字不及运动矢量增加的码字，再提高运动矢量精度反而会降低编码效率。德国学者 Girod 对这个问题进行了实验性研究，得到如下两个很有指导性的结论。

（1）简单的双线性插值效果与更复杂的 sinc 插值效果非常接近。

（2）对于广播电视图像 1/4 精度，对于会议电视图像 1/2 精度是比较合适的。

在已有的序列图像编码标准中，MPEG 和 H.263 均采用了 1/2 精度运动估计，均证实了比整数运动估计更有效。

下面以 1/2 精度为例，介绍分数精度运动估计的基本方法，其他分数精度算法只是其简单的推广。

在 1/2 精度特例下，插值公式只需得到位于 $X/2$ 和 $Y/2$ 格点上的像素值，如图 2.36 所示，×表示整像素，O 表示 1/2 位置像素，插值公式简化为：

$$\begin{cases} b = \dfrac{A+B}{2} \\ c = \dfrac{A+C}{2} \\ d = \dfrac{A+B+C+D}{4} \end{cases} \tag{2.71}$$

对前一帧图像插值，形成按 *X*/2，*Y*/2 为距离的像素阵，当前图像并不插值，插值像素示意图如图 2.36 所示。

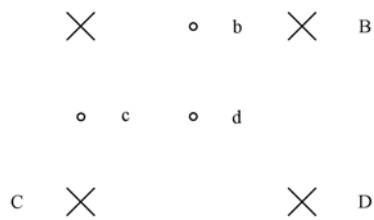


图 2.36　插值像素示意图

就图 2.37 的示意图，对匹配准则进行一些修改，以 MAD 准则为例，块 $B(I,J,k)$ 的 MAD 度量为：

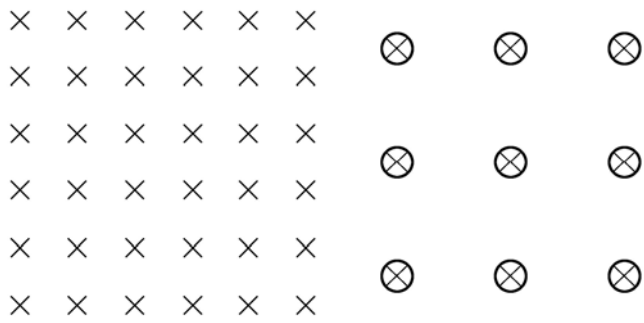


图 2.37　*k*−1 帧与 *k* 帧像素分布示意图

$$\begin{aligned} \mathrm{MAD}_2(i,j) = \frac{1}{MN}\sum_{n=0}^{N-1}\sum_{m=0}^{M-1}\big|S(I+n,J+m,k) - \\ \tilde{S}(2I+2n+i,2J+2m+j,k-1)\big| \end{aligned} \tag{2.72}$$

（*i*,*j*）是在运动搜索范围内取值的，是对搜索中心的偏移，在求和项内，对于前一帧的插值场 $\tilde{s}(i,j,k-1)$ 的坐标求和要乘以 2，但（*i*,*j*）因子不能乘 2，这样才能在 *X*/2，*X*/2 精度内进行全搜索，MSE_2 可类似地改写，以 $\mathrm{Crit}_2(i,j)$ 表示一般匹配准则。一个全搜索的 1/2 精度运动估计算法描述如下。

令 $d_{\max}$ 表示最大整数运动范围，相应的 1/2 精度搜索范围在 $\tilde{s}(i,j,k-1)$ 内为 $2d_{\max}+1$，计算如下位置的所有 $\mathrm{Crit}_2(i,j)$：

$$-2d_{\max}-1 \leqslant i \leqslant 2d_{\max}+1, \quad -2d_{\max}-1 \leqslant j \leqslant 2d_{\max}+1$$

并得到其最小值：

$$\mathrm{Crit}_2(i_0,j_0)=\min\left\{\mathrm{Crit}_2(i,j)\ \ -2d_{\max}-1\leqslant i\leqslant 2d_{\max}+1,\quad -2d_{\max}-1\leqslant j\leqslant 2d_{\max}+1\right\}$$

这样 $(i_0,j_0)^{\mathrm{T}}$ 就是所得的运动矢量，其中运动矢量整数部分为 $(i_0/2,j_0/2)^{\mathrm{T}}$，分数部分为 $(i_0\%,j_0\%)^{\mathrm{T}}$，这里%为取余数运算。

对于最大整数 $d_{\max}$ 的 1/2 精度运动估计，全搜索算法的总搜索次数为：

$$\left[2(2d_{\max}+1)+1\right]^2\approx 4\left(2d_{\max}+1\right)^2$$

这个搜索次数是整数全搜索的 4 倍多。

一个更实用的两步法，可以大大降低运算次数，描述如下。

（1）直接进行整数运动估计，得到整数运动矢量为 $(i_1,j_1)^{\mathrm{T}}$。

（2）将前一帧进行插值后，进行 1/2 运动估计，具体做法是在前一帧插值形成的图像内以 $(2i_1,2j_1)^{\mathrm{T}}$ 为搜索中心，以±1 为搜索区域，得到：

$$\mathrm{Crit}_2(2i_1+i_0,2j_1+j_0)=\min\left\{\mathrm{Crit}_2(2i_1+i,2j_1+j),-1\leqslant i\leqslant 1,-1\leqslant j\leqslant 1\right\}$$

$(i_0,j_0)^{\mathrm{T}}$ 为所得运动矢量的分数部分。

这个两步算法比整数像素运动矢量只多 8 次搜索，为了在名词上区别于三步法，称这个算法为局域 1/2 运动估计方法，还可以与整数快速搜索算法及多层搜索算法结合，得到更快速的实现。

获得运动矢量 $\mathbf{mv}=(\mathrm{dx},\mathrm{dy})^{\mathrm{T}}$ 后，块 $B(I,J,k)$ 的运动补偿预测值为：

$$\mathrm{MCP}(I+i,J+j,k)=\tilde{S}(2I+2i+\mathrm{dx},2J+2j+\mathrm{dy},k-1) \tag{2.73}$$

位移帧间差值为：

$$\mathrm{DFD}(I+i,J+j,k)=S(I+i,J+j,k)-\mathrm{MCP}(I+i,J+j,k) \tag{2.74}$$

2.7.4　重叠运动补偿预测（OMCP）

我们来观察一下以运动补偿为基础的序列图像（视频）压缩编解码器的工作过程，从中找出需要进一步解决的问题。

对于某一帧 k，分块后，估计每一块的运动矢量，得到每一块中所有像素的运动补偿预测值，从原像素值减去预测值得到位移帧间差值场 DFD，对运动矢量和 DFD 场进行编码，其中运动矢量的编码是无损编码，DFD 场的编码一般是有损编码，一般采用变换、量化和熵编码等过程，在解码器端，是一个反过程，这一正、反过程的数字描述如下。

对于任意点 (i,j)，设它的运动矢量 $\mathbf{mv}$ 为 $(\mathrm{dx},\mathrm{dy})^{\mathrm{T}}$，则：

$$\mathrm{MCP}(i,j,k)=S(i+\mathrm{dx},j+\mathrm{dy},k-1) \tag{2.75}$$

DFD 场值用 $e(i,j,k)$ 表示如下：

$$e(i,j,k)=S(i,j,k)-\mathrm{MCP}(i,j,k) \tag{2.76}$$

在解码端，恢复的是 DFD 的重构值，与原值存在一定的误差，用 $\hat{e}(i,j,k)$ 表示，则重构图像为：

$$\begin{aligned}\hat{S}(i,j,k)&=\hat{e}(i,j,k)+\mathrm{MCP}(i,j,k)\\&=\hat{e}(i,j,k)+S(d+\mathrm{dx},j+\mathrm{dy},k-1)\end{aligned} \tag{2.77}$$

从式（2.75）～式（2.77）分析和实验都发现以下几个问题：

（1）由于各块的运动矢量之间存在不连续性，真实运动物体边界与块边界也不一致，因此由式（2.75）所得 MCP 场中，存在块边界的不连续性，使 $e(i,j,k)$ 中存在块边界的不连续性，即 DFD 场中存在边界不连续性。

（2）如果对 DFD 场按分块进行变换和量化，由于各块之间统计分布相差较大，量化误差的分布也极不一致，使得解码端得到的 $\hat{e}(i,j,k)$ 在各块之间的损失存在不一致性，再加上 $\mathrm{MCP}(i,j,k)$ 中的边界不连续性，由式（2.77）重构的图像，就存在块效应，随着码率越低，图像复杂性越高，块效应越明显。

（3）如果对 DFD 场整体进行变换和量化，由于块边界不连续性，使变换系数中高频分量增加，降低了 DFD 场的编码效率。

重叠运动补偿预测的目的是为了改善以上几个问题，对于一个块中的任意像素点（i,j），它们运动补偿预测不仅由本块运动矢量 $\mathbf{mv}^0=(\mathrm{dx}^0,\mathrm{dy}^0)^{\mathrm{T}}$ 获得，也受邻近块的运动矢量 $\mathbf{mv}^n=(\mathrm{dx}^n,\mathrm{dy}^n)^{\mathrm{T}}$ 的影响，是一个加权叠加：

$$\mathrm{OMCP}(i,j,k)=\sum_n W^n(i,j)\cdot S(i+\mathrm{dx}^n,j+\mathrm{dy}^n,k-1) \tag{2.78}$$

这里 n 的取值为 0,1,2,……要视取哪些邻近块而定，如图 2.38 所示，左图 n 取值为 0，1，2，…，7，右图 n 的取值为 0,1,2,3,4。

1	2	3
4	0	5
6	7	8

	1	
2	0	3
	4	

图 2.38　邻近窗的利用

不管式（2.78）中取几个邻近块，对于任意（i,j）总有 $\sum_n W^n(i,j)=1$。$W^n(i,j)$ 表示加权函数在（i,j）点的值，它实际上是一个窗函数，一般情况下，$W^0(i,j)$ 要占主导地位。

获得重叠运动补偿预测 OMCP 以后，类似地，DFD 场（用 $e(i,j,k)$ 表示）为：

$$e(i,j,k)=S(i,j,k)-\mathrm{OMCP}(i,j,k) \tag{2.79}$$

在解码端，得到重构的 DFD 场，图像帧恢复为：

$$\begin{aligned}\hat{S}(i,j,k)&=\hat{e}(i,j,k)+\mathrm{OMCP}(i,j,k)\\&=\hat{e}(i,j,k)+\sum_n W^n(i,j)S(i+\mathrm{dx}^n,j+\mathrm{dy}^n,k-1)\end{aligned} \tag{2.80}$$

由于重叠运动补偿的加窗平均特性，使得 OMCP 场内块边界不连续性大大降低，同样 DFD 块中块边界不连续性也大为降低，特别适合于对 DFD 场作整体变换的情况，即使是对 DFD 场作分块变换，块效应也得到相当的抑制，H.263 建议了一个重叠补偿的具体实现方案，并给出了一组加权矩阵（窗函数）。

2.7.5　双向预测

当前帧的图像可以从前一帧和后一帧实行双向预测，对于一个块，它对前一帧的运动矢

量为 $\mathbf{mv}_f$,对后一帧的运动矢量为 $\mathbf{mv}_b$，块内任一点（*i,j*）的双向预测为：

$$\begin{aligned}\mathrm{BMP}(i,j,k) = w_f S(i+\mathbf{mv}_f(x), j+\mathbf{mv}_f(y), k-1) + \\ w_b \cdot S(i+\mathbf{mv}_b(x), j+\mathbf{mv}_b(y), k+1)\end{aligned} \tag{2.81}$$

这里要求 $w_f + w_b = 1$。

假设仍使用块匹配方法预测双向运动矢量 $\mathbf{mv}_f$ 和 $\mathbf{mv}_b$,假设前向运动估计 $\mathbf{mv}_f$ 的搜索范围为 $d_{1\max}$，后向运动估计 $\mathbf{mv}_b$ 的搜索范围为 $d_{2\max}$，则使用双向搜索算法，共计需要搜索的次数为 C_{SB}。

$$C_{SB} = (2d_{1\max}+1)^2(2d_{2\max}+1)^2$$

若令 $d_{1\max} = d_{2\max} = d_{\max}$，则：

$$C_{SB} = (2d_{\max}+1)^4 \tag{2.82}$$

为了降低搜索次数，可以采用以下算法：

（1）令 $\mathbf{mv}_f=(0,0)^{\mathrm{T}}$，$\mathbf{mv}_b = (0,0)^{\mathrm{T}}$，计算 $\mathrm{Crit}_B(\mathbf{mv}_f, \mathbf{mv}_b)$，且令 $S_t = \mathrm{Crit}_B(\mathbf{mv}_f, \mathbf{mv}_b)$。

（2）令 $\mathbf{mv}_b$ 不变，只在 $d_{\max}$ 范围内对 $\mathbf{mv}_f$ 进行前向搜索，得到最佳匹配的 $\mathbf{mv}_f$ 及 $\mathrm{Crit}_B(\mathbf{mv}_f, \mathbf{mv}_b)$，若不是第一次进入（2），且 $\left|S_t - \mathrm{Crit}_B(\mathbf{mv}_f, \mathbf{mv}_b)\right| < \delta$，则算法停止，否则令 $S_A = \mathrm{Crit}_B(\mathbf{mv}_f, \mathbf{mv}_b)$，转到（3）。

（3）令 $\mathbf{mv}_f$ 不变，只在 $d_{\max}$ 范围内对 $\mathbf{mv}_b$ 进行搜索，得到最佳匹配的 $\mathbf{mv}_b$ 及 $\mathrm{Crit}_B(\mathbf{mv}_f, \mathbf{mv}_b)$，若 $\left|S_t - \mathrm{Crit}_B(\mathbf{mv}_f, \mathbf{mv}_b)\right| < \delta$，则算法结束，否则令 $S_t = \mathrm{Crit}_B(\mathbf{mv}_f, \mathbf{mv}_b)$，转到（2）。

在以上算法中，如果采用 MAD 准则，对于块 *B*（*I*，*J*，*k*），其 $\mathrm{Crit}_B(\mathbf{mv}_f, \mathbf{mv}_b)$ 的定义为：

$$\begin{aligned}\mathrm{Crit}_B(\mathbf{mv}_f, \mathbf{mv}_b) = \frac{1}{MN}\sum_{n=0}^{N-1}\sum_{m=0}^{M-1}\big|S(I+n, J+m, k) - \\ w_f S(I+n+\mathbf{mv}_f(x), J+m+\mathbf{mv}_f(y), k-1) - \\ w_b \cdot S(I+n+\mathbf{mv}_f(x), J+m+\mathbf{mv}_b(y), k+1)\big|\end{aligned} \tag{2.83}$$

算法描述中的δ为预先设置的门限，如果两次迭代之间，Crit 的改善不明显，则停止迭代。

除第一步外，若总迭代次数为 *n* 次，则搜索次数为：

$$C_{SB} = n(2d_{\max}+1)^2 \tag{2.84}$$

一般情况下，*n*<10，式（2.84）比式（2.82）小得多，第二步和第三步，也可以使用快速算法或多分层搜索算法，进一步改善运算效率。在 MPEG、H.263 标准中，均使用了双向运动预测，在那里称为 B 帧。

2.8　序列图像编码算法

如果将序列图像或视频图像的每一帧当做一个单独的静态图像看待，则变换编码可直接用于每一帧图像的编码。因为序列图像的相邻帧存在很强的相关性，利用这种相关性，可以构造用于序列图像的专门算法，一般比直接采用静态图像编码方法更有效。

最重要的一类序列图像编码方法是混合编码方法，它是DPCM和变换编码的混合，DPCM指的是帧间预测，利用估计得到的运动矢量，从前一帧预测当前帧，真实值减去预测值，得到预测误差，如果采用前述的块平移运动，这个预测误差也称为帧间位移差（DFD），这就是DPCM的含义；对预测误差场（或称为残差场）再采用变换编码，就构成混合编码方法。

如果以帧为单位，在混合编码中，有两类帧存在，一类采用帧内编码，即静态变换编码；另一类采用帧间编码，即混合编码帧；初始时，第一帧必须采用帧内编码，在后续的编码过程中，每隔一段时间对一帧进行帧内编码，为的是编辑定位方便，或防止传输过程的误码引起累计差错，其他帧采用帧间编码。

帧内编码常采用块变换编码，帧间编码一般也采用块结构，对每一块（称为当前块）通过参考帧进行运动估计，参考帧一般是它的前一帧，或前面某一帧，参考帧是已经编码完成的帧，得到当前块的运动矢量后，通过帧间预测，得到该块每个像素的预测值，实际值减预测值得到块内每个像素的预测误差，对预测误差块直接或再分成更小的块进行块变换，例如进行DCT，对变换系数进行量化，扫描和熵编码。图2.39给出了帧间编码的示意图。

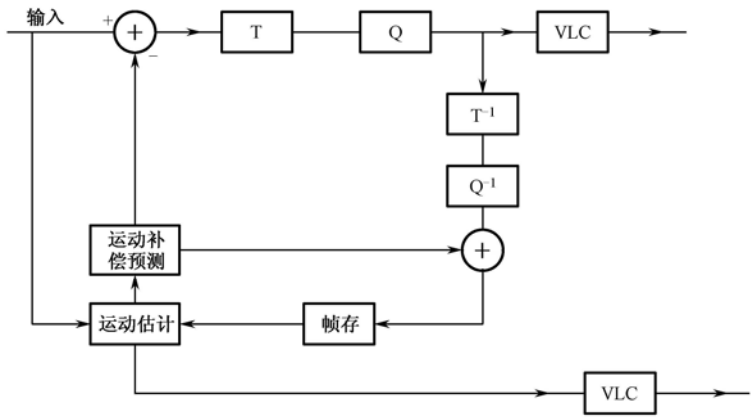


图2.39　帧间编码示意图

在图2.39中，帧存中存的是参考帧图像（一般是上一帧），输入当前图像帧分块后，取一个块作为输入，与帧存的一个搜索范围比较，估计运动矢量，由运动矢量得到该块的运动补偿预测值，输入减去该预测值得到预测误差，进行变换、量化和变长编码（熵编码）。注意，在这个图中，除VLD（变长解码）外，还包含一个完整的解码通道，这样做的目的是在帧存中保留一个解码后的有失真的前帧图像（对下一帧而言）作为参考帧，这一点与第2.5节中介绍的DPCM编码器中的解码环路目的是一致的，目的是为了在编码器端和解码器端得到一致的参考图像，避免误差积累。

因为变换编码和运动补偿技术分别在前几节详细讨论了，因此不再赘述。运动预测补偿和变换技术相结合构成序列图像最主要的压缩技术：混合编码，它们是构成当前已经公布的几种序列图像编码标准MPEG-1、MPEG-2、H.261、H.263和H.264的核心技术，也是MPEG-4中简单编码模式所采用的基本技术。

2.9　各种图像压缩标准的应用目标和主要技术

每一个图像压缩标准的制定，都瞄准它最合适的应用目标。H.261 是最早出现的视频编码标准，它首次使用了运动补偿预测编码加 DCT 变换的方法，其输出码率是 64 kb/s 的整数倍（1～31）。H.261 主要是针对 ISDN 的会议电视和可视电话等应用制定的，通过缓冲器控制产生恒定的输出码率。

与 H.261 不同，MPEG-1 标准不是针对通信领域的双向应用，而是为了视频存储媒体（如 VCD）的应用。该标准在典型的运动补偿预测编码（MCPC）框架的基础上，应用了半像素的双向预测技术，提供更好的编码质量和更高的压缩比，其主要应用目标在 1～1.5 Mb/s 的情况下，提供 25 帧 CIF（352×288）VHS 质量的图像。

MPEG-2 保留了 MPEG-1 的基本技术，并针对分场的图像格式，使用帧、场联合的编码结构，能够更好地支持高分辨率的图像和高质量音频编码。目标码率在 3～35 Mb/s 的传输速率条件下，提供目前广播级 TV 到 HDTV 的图像编码，同时，能够提供质量、时间、空间等多种可伸缩性的（分级）编码模式。MPEG-2 不像 MPEG-1 那样只是用于静态存储媒体，而是把应用目标扩展到单向通道的视频广播领域，比如卫星数字电视和有线电视信号。

考虑到双向的视频传输仍然是一个重要的应用，ITU 分析了 H.261 失败的原因，提出了 H.263 标准，主要面向低码率的视频应用，可以支持在电话线上传输可视电话和会议电视。

H.263 是为了支持低速率的通信而制定的标准，但希望能够适应较大的动态范围，不仅限于低码率。H263 对 CIF 图像，在 128 kb/s～1 Mb/s 码率范围内，一般可以获得比 MPEG-1 更好的压缩效果。

由于公用电话网（PSTN）和无线网络上的传输速率仍然有限，而且误码率较高，因此，人们又提出了 H.263 的改进版本，以满足高压缩效率和强信道容错能力的应用要求。H.263+以及后来的 H.263++能很好地解决低码率视频应用问题，它们在提高编码压缩效率的同时，提高了码流对有误码信道的容错能力。这些标准的容错能力，都是通过选项的形式实现的，方便灵活，并且能够兼容本标准的以前版本。H.263+是在 H.263 的基础上以增加编码可选项的形式改进的，在语法上与 H.263 兼容，但编码效率有很大提高，适用范围也更大。其主要的应用方向仍是低码率的视频业务，用于 PSTN 以及无线接入的有误码的通信环境，因此在 H.263+中既增加了一些改进编码效率的方法，同时也提高了抗误码性能的能力。由于实现成本较低，H263+标准已经越来越多地被采用。

H.263 系列标准具有里程碑的性质，在其中首次提出的很多概念，比如变块大小的运动估计、初始运动矢量预测、无限制运动估计、多参考帧运动补偿等都被其后的很多标准沿用。它们的压缩比也是相当出色的，其后一些标准的功能和码流的可操作性确实有所提高，但是压缩比和 H.263 系列相比并没有质的进步。

与 H.263 系列相对应，MPEG-4 标准也能够支持码率低于 64 kb/s 的视频应用，同时还能够支持广播级的视频应用。与其他压缩标准相比，MPEG-4 标准在块 DCT 编码的基础上扩展了视频对象的概念，并引入了基于对象、内容的编码技术，从而在理论上具有更高的压缩效率。但由于实际实现上目前仍存在的对复杂视频序列进行自动分割的困难，实时实现的 MPEG-4 编码器实际未能实现真正的面向对象的编码技术。在 MPEG-4 众多的档（Profile）

中，目前看来，最流行的也是最成功的是 Simple Profile 和 Advanced Simple Profile。前者基本和 H.263 类似，后者在 H.263 的基础上引入了 1/4 像素运动估计和全局运动估计技术，对象的概念和技术并未包含在这两个档里，因此，目前 MPEG-4 编码器在低码率应用条件下，仍具有和 H.263/H.263+大致相同的压缩能力。不过 MPEG-4 提供了一些 H.263 所没有的功能，比如 FGS 等，这使得它在有噪信道上的传输性能比 H.263 要好一些。

目前，ITU 和 ISO 联合制定的一个最新的视频编码标准是 H.264，在 ITU 的计划中，这个标准被称为 H.264，在 ISO 了里称为 MPEG-4－Part 10。

当前，静止图像压缩标准采用的主要技术是以变换编码为基础的混合编码方法，JPEG 标准使用块 DCT 变换和熵编码的结合, JPEG2000 使用基于小波变换的 EBCOT 编码方法，而所有运动图像编码技术均以块变换结合运动预测补偿的混合编码器为核心算法。表 2.13 对各标准所采用的主要技术和应用对象进行了概括。

表 2.13　各种编码标准的主要技术和应用目标

编 码 标 准	制 定 组 织	目 标 码 率	主要压缩技术	主要应用目标
JPEG	ISO/IEC	2～30 倍	DCT 主观量化 Zig-Zag 扫描 熵编码	Internet 图像服务 数字照相 图像和视频编辑
JPEG2000	ISO/IEC	2～50 倍	小波变换 EBCOT ROI 编码 空间可分级码流 质量可分级码流 改进算数编码 容错编码	Internet 图像服务 数字照相 图像和视频编辑 打印 医学图像 移动应用 彩色传真 卫星图像传输
MPEG-1	ISO/ICE	1.5 Mb/s	DCT 主观量化 自适应量化 Zig-Zag 扫描 熵编码 运动预测补偿 双向运动补偿 半像素运动估计	CD-ROM 视盘 消费视频 视频记录
MPEG-2	ISO/ICE	1.5 Mb/s～ 35 Mb/s	MPEG-1 所有技术 基于帧/场运动补偿 空间可分级码流 时间可分级码流 质量可分级码流 容错编码	数字 TV HDTV 高质量视频传输、存储 卫星 TV CATV DVB/DVD 视频编辑

续表

编码标准	制定组织	目标码率	主要压缩技术	主要应用目标
MPEG-4	ISO/ICE	8 kb/s～ 35 Mb/s	MPEG-2 所有技术 Wavelet 零树扫描 高级运动补偿 重叠运动补偿 视相关可扩展编码 位图形状编码 Sprite 编码 脸部动画 动态网格编码	Internet 交互视频 可视编辑 内容管理 消费视频 专业级视频 2D/3D 计算机图形 移动通信
H261	ITU-T	P×64 kb/s P:1-31	DCT 自适应量化 Zig-Zag 扫描 熵编码 运动预测补偿 整像素运动估计 差错控制编码	ISDN 视频会议
H263	ITU-T	8 kb/s～ 1.5 Mb/s	H.261 全部技术 双向运动补偿 半像素运动估计 高级运动补偿 重叠运动补偿 可选算数编码 无限制运动预测	可视电话 桌面可视电话 桌面电视会议 移动可视电话 网络视频
H263+ H263++	ITU-T	同上	增加 12 个选项 增加 4 个选项	同上
H.264	ITU-T ISO-ICE		帧内预测编码 新的运动估计方法 可变块大小 多帧运动估计 1/4 甚至 1/8 像素精确度运动估计 解码环路中的去块滤波器 整数 DCT 变换 新的熵编码方法 CAVLC（Context-based Adaptive Variable Length Coding） CABAC（Context-based Adaptive Binary Arithmetic Coding）	通过电缆、卫星、CableModem、DSL、陆地等媒介的广播 在光学或磁性设备、DVD 上的交互式储存 基于 ISDN、以太网、LAN、DSL、无线移动网络、调制解调器的交互服务 基于 ISDN、CableModem、DSL、LAN、无线网络的视频点播和多媒体流服务 基于 ISDN、DSL、以太网、LAN、无线和移动网络等的多媒体消息服务（MMS）

随着人们需求的提高，视频编码技术也会不断发展，但是就目前而言，从 H.264 的制定看，真正对于压缩比有提高的可行的方案，往往仍然是一些十分简单的技术，比如 H.264 中使用的变块大小运动估计技术等，这可能在很大程度上是要减少运算成本。

参 考 文 献

［1］张旭东，卢国栋，冯健．图像编码基础和小波压缩技术．北京：清华大学出版社，2004．

［2］朱雪龙．信息论基础，北京：清华大学出版社，2001．

［3］刘峰．视频图像编码技术及国际标准．北京：北京邮电大学出版社，2005．

［4］姚庆栋，毕厚杰，王兆华，徐孟侠．图像编码基础（第三版），北京：清华大学出版社，2006．

第 3 章　TMS320C6000 实现 JPEG 编解码器

本章讨论在 TMS320C6000 的 DSP 上实现 JPEG 编解码器的问题。我们首先较详细地介绍 JPEG 标准，然后概要介绍 JPEG 在 TMS320C6000 DSP 上实现的一些技术问题。

3.1　JPEG 编码标准

JPEG 是第一个被广泛接受的单色和彩色静止图像压缩标准，它的名字源于“Joint Photographic Experts Group”，它是由 ISO 和 CCITT 协同工作的机构，这个工作的结果是 ISO 的国际标准 ISO/IEC 10918-1（连续色调静止图像的数字压缩和编码，Digital Compression and Coding of Continuous Tone Still Images）和 ITU-T 的建议 T.81（CCITT 是 ITU 的前身）。JPEG 标准草案于 1991 年公布，1992 年正式批准为国际标准，之后这个工作进一步增强和扩展形成 ISO 10918-3 和 ITU-T 建议 T.84。

国际标准的目标是提供一个“可互相工作和访问”的工具，即不同生产商提供的芯片或软件产生的压缩码流是互相可解码的，这为商业应用提供一个基准。目前，JPEG 已广泛应用于与图像压缩相关联的应用中，如图像数据库、新闻图片传输、桌面印刷系统、医学图像、卫星遥感图像传输、数码照相机等。

作为一个通用的图像压缩标准，JPEG 的制定满足几个原则，首先要反映当时先进的图像压缩算法的水平，其次要在压缩比、图像质量、运算复杂性、软/硬件实现的结构有效性等方面折中。还要满足通用性原则，一方面要适应各种图像类型，如人脸、建筑、自然景物、医学成像等；另一方面要适应各种彩色空间，图像大小和分辨率等。标准还应该提供各种工作模式，以适应不同的应用要求，如有失真、无失真、顺序工作方式、渐进方式、多种分辨方式等。JPEG 的制定，基本上满足了这些原则。JPEG 中的核心算法是 DCT 变换编码，其压缩性能基本反映了 20 世纪 80 年代末期图像压缩的技术水平。但自从 JPEG 制定后的近 10 年，许多更有效的图像压缩技术已得到发展，如小波变换方法、分形方法、区域划分方法等。在这其中，发展最成熟和性能及通用性最好的静止图像压缩方法是小波变换方法，正是这种背景，第二代静止图像压缩标准已经制定，标准文本于 2000 年公布，这就是 JPEG2000，它的核心技术正是小波变换编码。

本节概要介绍 JPEG 标准的主要算法，并通过一些例子，说明它的压缩性能，在此基础上介绍 JPEG 在 DSP 上实现的基本流程。

3.1.1　JPEG 标准的工作模式

为了适用于单色或彩色图像，JPEG 对每一个图像分量单独编码，对单色图像，它只有一个分量，反映图像的亮度。对彩色图像，一个图像分量是指一种彩色分量。例如，RGB 彩

色图像，R，G，B 分别构成三个图像分量单独编码，对 YUV 彩色图像，亮度 Y 是单独分量，色差 U 和 V 各构成一个单独图像分量，但各图像分量可能用不同的分辨率，例如，Y 用 512×512 的图像尺寸，而 U 和 V 分量可以采用 256×256 的尺寸。在 JPEG 标准中，可以用参数 X 和 Y 表示最大图像分量尺寸为 $X\times Y$，而用 H_i 和 V_i 表示每个分量宽度和高度的相对大小，每个分量的实际尺寸 X_i 和 Y_i 可以通过下式计算：

$$X_i = XH_i / H_{\max},\quad Y_i = YV_i / V_{\max}$$

式中 $H_{\max} = \max\{H_i, i = 1,2,\mathrm{L}\ ,n\}, V_{\max} = \max\{V_i, i = 1,2,\mathrm{L}\ ,n\}$，图像分量的个数不超过 256 个，$H_i$ 和 V_i 取值为 1～4 之间的整数。

JPEG 对每个不同的图像分量可以采用不同的量化参数和熵编码的码表，JPEG 本身并不进行分量间的转换，如果对 RGB 彩色图像转换成 YUV 分量再进行压缩，这个转换要由 JPEG 以外的功能单元完成，JPEG 本身不进行这种转换。

有了以上讨论，在下面的论述时，我们仅针对单分量图像，只有在必要时，才会提到亮度和彩色的一些不同处理方式。

对于一个图像分量，JPEG 提供 4 种工作模式。

- 顺序编码：单遍扫描完成一个图像分量的编码，扫描次序从左到右，从上到下。
- 渐进编码：通过多遍扫描完成一个图像分量的编码，每一遍扫描使图像质量更好，这种方式用于当传输信道非常慢时，它提供一个由粗到精的渐进码流结构。
- 无失真编码：通过预测编码方式，提供一个无失真的编码模式。
- 分层编码：提供多分辨率的码流结构。

在中等复杂度的图像情况下，JPEG 的有失真编码可以达到以下的码率和性能。

- 0.25～0.5 bpp，中等至较好图像质量，适用于一定的应用；
- 0.5～0.75 bpp，较好至好的图像质量，满足许多应用；
- 0.75～1.5 bpp，非常好的图像质量，满足大多数应用；
- 1.5～2.0 bpp，与原始图像几乎不可分辨差别，几乎满足所有应用。

在无失真压缩情况下，可达到约 2∶1 的压缩比。

3.1.2　基本工作模式

在 JPEG 应用中，最常用的工作方式是顺序编码中的基本工作模式，实际上，许多硬件 JPEG 编码器和商用 JPEG 软件仅支持这种工作模式，本节以基本工作模式（Baseline Mode）为例，介绍 JPEG 的主要算法，各种增强和扩展在下一小节介绍。

JPEG 基本工作模式下，编码器和解码器的方框图如图 3.1 所示。

在基本工作模式下，输入图像限制为 8 位，设 p=8，图像取值范围限定在 $0\text{–}2^p\text{–}1$ 的无符号的整数，将像素值移位 2^{p-1}，形成 $-2^{p-1}\text{–}2^{p-1}\text{–}1$ 范围内的有符号整数。将图像分成 8×8 的小块，每个块按从左到右，从上到下顺序送入 DCT 变换器（即前向 DCT，FDCT）进行二维 DCT 变换，块编号用 B 表示。块变换系数表示为 $y_B(k, l)$，变换系数送入量化器，由量化器输出量化值的序号。量化器是一个均匀量化器，但每个变换系数的量化步长是不一致的。JPEG 标准并没有规定一个固定的量化步长矩阵，但却给出一个量化步长矩阵的例子，可以用默认值，在灰度图像情况下的亮度量化矩阵如下所示。

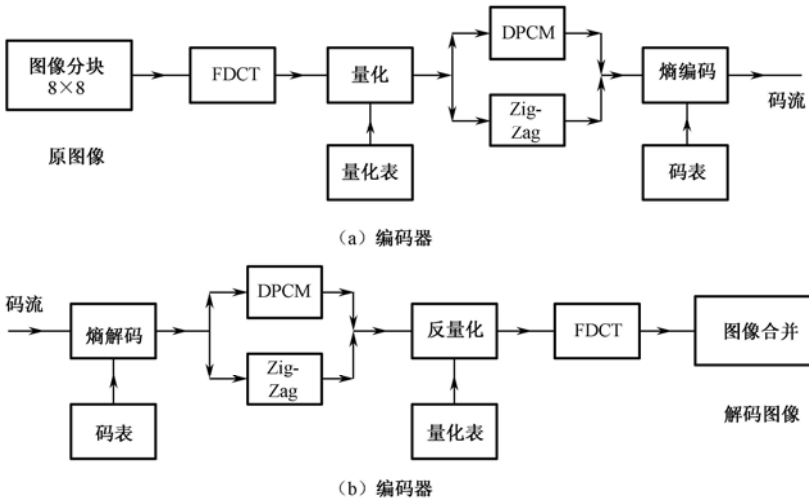


图 3.1　JPEG 工作模式的编码器和解码器

$$\begin{bmatrix} 16 & 11 & 10 & 16 & 24 & 40 & 51 & 61 \\ 12 & 12 & 14 & 19 & 26 & 58 & 60 & 55 \\ 14 & 13 & 16 & 24 & 40 & 57 & 69 & 56 \\ 14 & 17 & 22 & 29 & 51 & 87 & 80 & 62 \\ 18 & 22 & 37 & 56 & 68 & 109 & 103 & 77 \\ 24 & 35 & 55 & 64 & 81 & 104 & 113 & 92 \\ 49 & 64 & 78 & 87 & 103 & 121 & 120 & 101 \\ 72 & 92 & 95 & 98 & 112 & 100 & 103 & 99 \end{bmatrix}$$

在彩色 YUV 格式下，亮度分量的量化矩阵与灰度图像一致，色度的量化矩阵如下所示。

$$\begin{bmatrix} 17 & 18 & 24 & 47 & 99 & 99 & 99 & 99 \\ 18 & 21 & 26 & 66 & 99 & 99 & 99 & 99 \\ 24 & 26 & 56 & 99 & 99 & 99 & 99 & 99 \\ 47 & 66 & 99 & 99 & 99 & 99 & 99 & 99 \\ 99 & 99 & 99 & 99 & 99 & 99 & 99 & 99 \\ 99 & 99 & 99 & 99 & 99 & 99 & 99 & 99 \\ 99 & 99 & 99 & 99 & 99 & 99 & 99 & 99 \\ 99 & 99 & 99 & 99 & 99 & 99 & 99 & 99 \end{bmatrix}$$

实际的量化步长可以用一个比例因子乘上述的量化矩阵，得到对一个应用合适的量化步长，每个系数量化步长记为 $Q(k,l)$，量化器输出：

$$y_B^Q(k,l) = \text{round}\frac{y_B(k,l)}{Q(k,l)}$$

这里 round 是舍入运算，输出到最接近的整数，在解码器中反量化用于重构变换的系数为：

$$\hat{y}_B(k,l) = y_B^Q(k,l)Q(k,l)$$

这是一个典型均匀量化器，量化区间是 $[(I-0.5)Q,(I+0.5)Q]$，量化区间的代表值为 IQ，量化器输出序号为 I，这里 $I=0,\pm1,\pm2,\cdots\cdots$

量化器输出的值，按直流系数 DC 和交流系数 AC 分成两类处理，对直流系数用 DPCM 编码，然后形成一个符号，以进行熵编码，当前块 B 的量化直流系数 $y_B^Q(0,0)$ 减上一块的直

流系数 $y^{Q}_{B-1}(0,0)$，得到差 DC_B。

$$DC_B = y^{Q}_{B}(0,0) - y^{Q}_{B-1}(0,0)$$

对 DC_B 进行编码，而不是直接处理 $y^{Q}_{B}(0,0)$，这样，可以进一步去除块间平均值之间的相关性。

由于输入图像取值范围为$-2^7 \sim 2^7-1$，$y^{Q}_{B}(k,l)$ 的取值范围为$-2^{10} \sim 2^{10}-1$，DC_B 的取值范围为$-2^{11} \sim 2^{11}-1$，如果用固定长码字表示，需 12 位，JPEG 给出另一种表示，将 DC_B 表示分成两个符号，第一个符号表示 DC_B 的取值范围和后续符号需要的附加码长，记为 SIZE；第二个符号是附加码长，用于真实表示 DC_B 的值，记为 AMP，DC_B 形成以下两个符号：

符号 1　　符号 2

（SIZE）　　(AMP)

符号 1 的取值和所代表符号 2 的取值范围如表 3.1 所示。

表 3.1　DC_B 符号 1 的取值及表示的范围

取　值	表 示 范 围
0	0
1	−1,1
2	−3, −2,2,3
3	−7,…, −4,4,…,7
4	−15,…, −8,8,…,15
5	−31,…, −16,16,…,31
6	−63,…, −32,32,…,63
7	127,…, −64,64,…,127
8	−255,…, −128,128,…,255
9	−511,…, −256,256,…,511
10	−1 023,…, −512,512,…,1 023
11	−2 047,…, −1 024,1 024,…,2 047

接下来，DC_B 的两个符号 SIZE 和 AMP 送入熵编码器，形成最后的码字，只有符号 SIZE 进行 Huffman 编码，AMP 则用 SIZE 位二进制表示，在 JPEG 标准中，可以由用户提供 SIZE 和 Huffman 码表，也可以采用 JPEG 提供的一个码表，对于亮度和色度分量，JPEG 提供的 Huffman 码表是不一致的，表 3.2 列出了用于表示 SIZE 的 Huffman 码表。

表 3.2　SIZE 的 Huffman 码表

	亮度直流系数		色度直流系数	
序　号	码　长	码　字	码　长	码　字
0	2	00	2	00
1	3	010	2	01
2	3	011	2	10
3	3	100	3	110
4	3	101	4	1110

续表

	亮度直流系数		色度直流系数	
序　号	码　长	码　字	码　长	码　字
5	3	110	5	11110
6	4	1110	6	111110
7	5	11110	7	1111110
8	6	111110	8	11111110
9	7	1111110	9	111111110
10	8	11111110	10	1111111110
11	9	111111110	11	1111111110

接下来，符号 2（即 AMP）用 SIZE 位二进制表示，表示方法是：如果 AMP>0，直接用 SIZE 位二进制原码表示，如果 AMP<0，用 AMP 的 1 的补码表示，即先得到|AMP|的 SIZE 位二进制表示，再各位取反，例如，DC_B=15，查找表 3.1 得到 SIZE=4，查找表 3.2 得到它的 Huffman 码字是 101，AMP 的二进制表示是 1111，因此，最终码字是 101，1111；如果 DC_B=−15，SIZE=4，AMP=−15，SIZE 的 Huffman 码字仍为 101，但 AMP 的编码为 0000，最终码字为 101，0000。

下面讨论 63 个 AC 系数的编码，对于 63 个 AC 系数，按 Zig-Zag 扫描次序（不包括 DC 系数）形成一个数字串，通过这个数字串，形成下列两个符号组成的符号表：

符号 1　　符号 2

（RUN，SIZE）　　（AMP）

每个非零系数对应于上面两个符号，符号 1 包括 RUN 和 SIZE 两个数字，RUN 表示当前非零值和前一个非零值之间 0 的个数，RUN 取值为 0～15，SIZE 表示用于刻画 AMP 所需要的位数和 AMP 所处的范围，SIZE 和非零 AC 系数取值范围的关系如表 3.3 所示。

表 3.3　SIZE 和非零 AC 系数取值范围的关系

SJZE 的取值	AC 系数范围
1	−1,1
2	−3, −2,2,3
3	−7,…, −4,4…,7
4	−15,…, −8,8,…,15
5	−31,…,−16,16,…,31
6	−63,…, −32,32,…,63
7	−127,…, −64,64,…,127
8	−255,…, −128,128,…,255
9	−511,…, −256,256,…,511
10	−1 023,…, −512,512,…,1 023

在形成 AC 系数符号表时，有两个特殊符号，EOB 表示已经遇到块中最后一个非零系数，ZRL=（RUN，SIZE）=（15，0）是另一个特殊符，它用于表示大于等于 16 的零行程，如果连续 0 的个数等于或超过 16，用 ZRL 表示 16 个连续的零，后面的符号中，RUN 部分相应减 16，ZRL 可以连续用于指示大于 16/32/48 的情况。符号 1（RUN，SIZE）用一个二维 Huffman 码表进行编码，给出 RUN 和 SIZE 的值，从二维 Huffman 码表中查出相应码字。同样，JPEG

允许用户定义自己的二维 Huffman 码表，它也给出两个默认的码表，分别用于编码亮度分量和色度分量，这两个码表可查阅 JPEG 文档或有关参考文献，AMP 的编码同 DC_B 一致。

下面给出一个例子，说明 JPEG 过程，它是 Lena 测试图像（分辨率 256×256），从 72×72 开始的一个块，它的前一个块的量化 DC 系数为−10，这个块取值如下：

107	105	104	114	100	112	111	108
104	99	107	108	112	115	117	115
104	101	108	110	109	114	117	114
105	105	105	106	110	109	96	113
102	107	102	113	105	104	107	115
107	106	102	103	106	115	106	121
114	107	87	98	110	102	116	120
114	99	98	95	93	111	115	112

每像素值减 128 后，进行 DCT 变换，然后进行量化，量化器输出为：

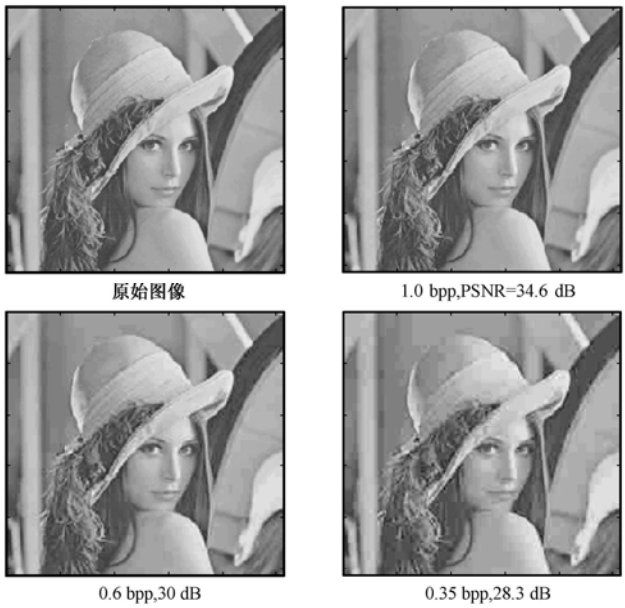
−1	−2	1	0	0	0	0	0
1	0	−1	0	0	0	0	0
0	0	1	0	0	0	0	0
0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0

可以形成这个块的实际码字，DC_B=−10−1=−11，SIZE=4，AMP=−11，编码为 101，0100，Zig-Zag 扫描为−2，1，0，0，1，0，−1，0，0，0，0，1，EOB。形成[RUN，SIZE][AMP]串为 [0，2][−2]，[0，1][1]，[2，1][1]，[1，1][−1]，[4，1][1]，[EOB]。对[RUN，SIZE]查 JPEG 文档的 Huffman 码表，对 AMP 直接编码，得到的码字为[01][01]，[00][1]，[11100][1]，[1100][0]，[111011][1]，[1010]。DC 编码需要 7 位，AC 编码需要 29 位，共需要 36 位，压缩比为 64×8/36=14.2。用解码器解码后，这个块的重构图像为：

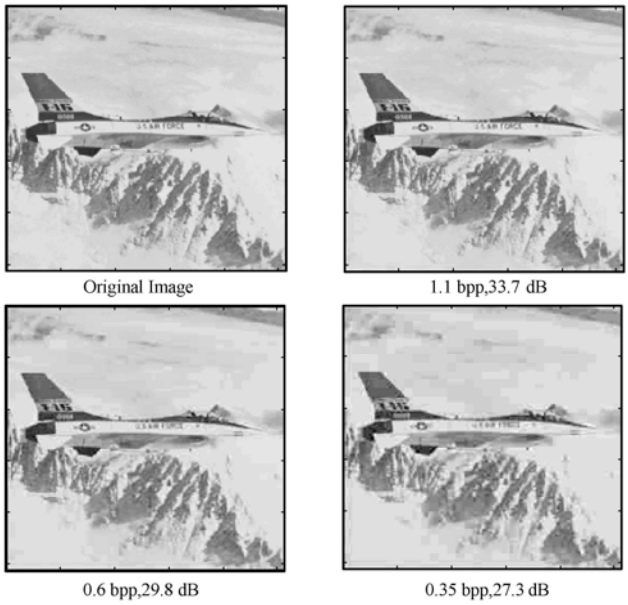
108	108	107	107	109	111	114	116
106	107	107	109	110	112	113	114
104	105	108	110	112	112	112	111
102	104	107	110	112	112	111	110
103	104	106	108	109	110	110	110
105	104	104	104	106	108	111	113
108	105	102	100	101	106	112	116
110	106	100	97	98	105	113	118

均方误差为 23.78，峰值信噪比为 34.4 dB。

图 3.2 给出在几个不同码率下，JPEG 对两个 256×256 测试图像的压缩效果，图 3.2（a）是 Lena 亮度图像的结果，图 3.2（b）是 Airplane 测试图像的压缩结果，在 0.35 bpp 情况下，块效应已经明显了。



（a）Lena 测试图像的压缩效果



（b）Airplane 的压缩效果

图 3.2　JPEG 压缩实例

3.1.3　其他工作模式

1）顺序编码

在顺序编码时，除基本模式外，还有其他选择方式，例如，可以支持 12 位像素精度，在这种情况下，SIZE 的类型增加 4，还可以选择算术编码替代 Huffman 编码。

2）渐进工作方式

在渐进工作方式下，通过多次扫描，得到由粗到精的图像质量的码流表示。渐进工作方式需要一个对整个图像进行 8×8 块变换和量化后的量化系数进行缓存的缓存器，对每一个块，首先选择一部分系数或部分精度进行熵编码和传输，解码端得到一个非常粗的重构图像，每次扫描增加一部分系数或精度，产生使图像质量得以改善的附加码流，这个过程重复几次，最终达到要求的质量。

渐进工作方式有两种渐进方式：谱选择和逐次逼近。谱选择是在每次扫描时，选择部分变换系数（按 Zig-Zag 扫描次序，从低频向高频），逐次逼近是先传输 l_1 个最高位，下次扫描再选 l_2 个次高位，依次下去，直到全部有效位得以传输。两种方式可以任意地组合，图 3.3 给出两种方式的示意图，图 3.4 是一个实际图像采用谱选择方式渐进编码的结果，第 1 遍扫描仅选择 DC 系数，第 2 遍扫描选择次 2 个最低频系数，第 3 遍扫描再选择次 5 个低频系数，最后一遍扫描取剩余系数。

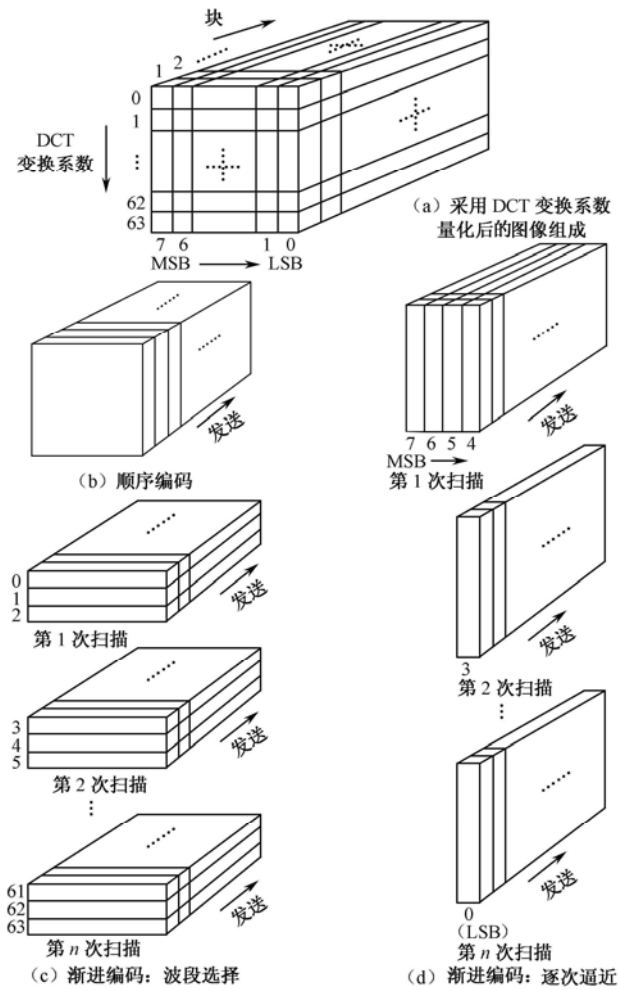


图 3.3　两种渐进扫描工作方式

图 3.4　渐进扫描编码实例

3）分层工作方式

分层工作方式用于提供多分辨码流结构，对原始图像进行低通滤波和 2∶1 采样（水平或垂直），得到在水平和（或）垂直方向 1/2 大小的图像，这个过程一直进行下去，得到 L 层次图像 $\{x_l(i,j), l=0,1,\mathrm{L}\ ,L-1\}$。编码时，首先编码 $X_{L-1}(i,j)$，可以用 JPEG 的基本模式，得到码流 C_{L-1}，对 C_{L-1} 解码得到 $X_{L-1}(i,j)$ 的解码图像 $\hat{x}_{L-1}(i,j)$，对 $\hat{x}_{L-1}(i,j)$ 在水平和垂直方向进行线性插值，得到两倍大小的图像 $\hat{x}^{p}_{L-1}(i,j)$，接下来对 $x_{L-2}(i,j)-\hat{x}^{p}_{L-1}(i,j)$ 进行编码，得到码流 C_{L-2}，这个过程一直进行下去，得到码流组 C_{L-1}，C_{L-2}，…，C_0，如果在传输和显示过程中，只使用 C_{L-1}，则得到 $1/2^{L-1}$ 原图像大小的解码图像，如果使用 C_{L-1}，C_{L-2}，则得到 $1/2^{L-2}$ 原图像大小的解码图像，一直到使用整个码组，得到原图像大小的解码图像，由此得到一个多分辨率码流组。

4）无失真编码

JPEG 提供了 8 种预测方式，通过一个选择器选择，这 8 种模式见表 3.4。

表 3.4　8 种选择模式

选择器	预测
0	无预测
1	*A*
2	*B*
3	*C*
4	*A*+*B*−*C*
5	*A*+(*B*−*C*)/2
6	*B*+(*A*−*C*)/2
7	(*A*+*B*)/2

各像素分布关系如图 3.5 所示。

C　　*B*

A　　*X*

图 3.5　像素分布关系

在无失真压缩方式下，输入图像可以是 2～16 位精度像素，可选用预测器 1～7，预测误差采用熵编码，如果熵编码采用 Huffman 编码，则预测误差编码方式类似于基本模式下的 DC 系数编码，熵编码也可以采用算术编码。

3.2　JPEG 在 C6000 上的实现

在已经了解 JPEG 算法原理的基础上，本节概要说明在 DSP 上实现一种 JPEG 编码器的主要技术流程，本节根据 TI 应用文档编译，该文档描述了 JPEG 在 TMS320C6000 DSP 上对静止图像压缩的实现情况。这一 JPEG 实现受以下条件约束：

- 基于离散余弦变换的 8 位精度的像素样本（Y-Cb-Cr　4∶4∶4/4∶2∶2/4∶2∶0）。
- 两个量化表格（分别对应亮度和色度），支持标准文档中的默认表格，也支持任意表格，即每张表格都可能由用户更改。
- 两个直流和两个交流表格（对亮度和色度分开设置），支持在标准文档中的表格 K3，K4，K5 和 K6，只限于支持这些表格。
- 编码器需要图像数据在存储器中有三个分开的按光栅扫描格式存放的图像分量（不支持隔行 Y-Cb-Cr）。
- 解码器将图像输出到相邻的三个图像分量中，输出图像按逐行光栅扫描方式存储。
- 每次扫描包含一个完整的图像分量，一次扫描只包含一个图像分量。
- 即使重启间隔的存在不会影响解码过程，该实现也不对重启间隔进行处理。
- 解码器需要第一次扫描从比特流中的第一个 DMA（直接存储器存取）数据包开始，第一个 DMA 数据包至少要包括以下标记：

```
0xFFD8:    图像起始;
0xFFC0:    一帧起始 (Baseline DCT);
0xFFDB:    量化表定义（Define Quantization Table）;
0xFFDA:    扫描起始（Start of Scan）。
```

因为 JPEG 编码器在前 590 字节以内输出以上内容，所以解码器的 DMA 数据包输入最小为 590 字节，而默认值是 640 字节。

- 不支持无损 JPEG 编码/解码；
- 不支持渐进图像传输功能；
- 像素的尺寸（行，列）必须是 8 的倍数；
- 编码器提供简单的压缩率控制功能；
- 解码器只能对那些与本编码器产生的有相同结构的比特流进行解码。

JPEG 的基本原理已如上节所述，其技术细节请参考 ISO DIS 10918 的 JPEG 标准文档。

3.2.1　JPEG 编码器

1. JPEG 编码器算法

图 3.6 为 JPEG 编码器处理过程的总体流程。

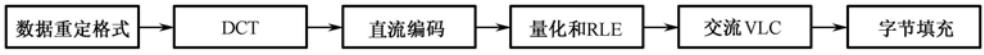
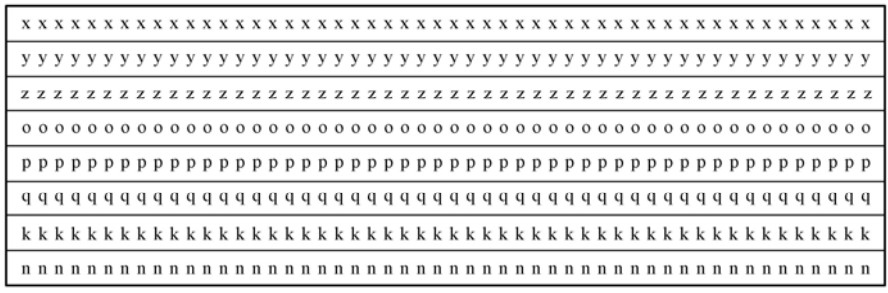


图 3.6　JPEG 编码器

下面分别对几部分进行概要说明。

（1）数据重定格式：这一操作将光栅扫描的图像数据转换成为彼此相邻的 8×8 大小的图像块。图 3.7 为存储在内存中的原始数据，图像结构中在同一行的所有图像样本用一个字母表示，图 3.8 显示了基于压缩方案的图像块所需要的重定格式的数据。这一操作同时也将像素亮度的动态范围从（0～255）调整到了（−128，127），从而消除了信号中的直流分量。


x x x

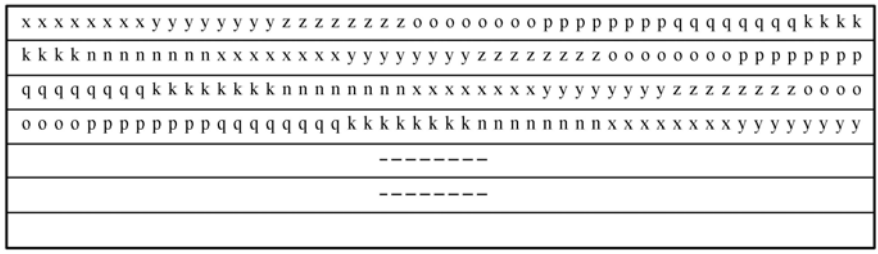


图 3.8　重定格式的图像数据

（2）DCT：这一操作对重定格式后的 8×8 图像数据块进行 2D 离散余弦变换，输出相应的 2D 频率成分 8×8 数据块。

$$S_{vu}=\frac{1}{4}C_uC_v\sum_{x=0}^{7}\sum_{y=0}^{7}S_{yx}\cos\frac{(2x+1)u\pi}{16}\cos\frac{(2y+1)u\pi}{16}$$

其中，当 u,v=0 时，$C_u,C_v=1/\sqrt{2}$；否则，$C_u,C_v=1$，S_{vu} 是在 u,v 处的 DCT 成分，S_{yx} 是图像像素在 x,y 处的空间样本值。

如下所述，将 2D 离散余弦变换分成两个 1D 操作来减小操作处理的数量。

- 完成 8 个 1D DCT，每一个都对应排列中的行。
- 完成 8 个 1D DCT，每一个都对应排列中的列。

（3）直流编码：这一步骤量化直流系数并对其进行 Huffman 编码。直流系数在 JPEG 中是差分编码的，前后 DC 分量之差被计算、量化和编码。在执行这一步骤时，使用到了量化器的预先计算功能（一个倒数量化表被预先计算和存储）。

（4）量化和 RLE：这一步骤量化交流系数，用 Zig-Zag 扫描和进行行程编码。在执行这一步骤时，使用到了量化器的预先计算功能（预先计算和存储一个倒数量化表）。Zig-Zag 扫描变换重置系数如图 3.9 所示。

0	1	2	3	4	5	6	7
8	9	10	11	12	13	14	15
16	17	18	19	20	21	22	23
24	25	26	27	28	29	30	31
32	33	34	35	36	37	38	39
40	41	42	43	44	45	46	47
48	49	50	51	52	53	54	55
56	57	58	59	60	61	62	63

0	1	5	6	14	15	27	28
2	4	7	13	16	26	29	42
3	8	12	17	25	30	41	43
9	11	18	24	31	40	44	53
10	19	23	32	39	45	52	54
20	22	33	38	46	51	55	60
21	34	37	47	50	56	59	61
35	36	48	49	57	58	62	63

图 3.9　Zig-Zag 扫描变换重置系数（输入和输出）

（5）交流 VLC：这一步骤完成变长编码（VLC），变长编码通过查 Huffman 码表实现。

（6）字节填充：在 JPEG 标准中，控制标志字为 0xFF，这一标志后面接着一个或多个控制指令，但在 0xFF 字节后面如果接着 0x00 字节，表示此 0xFF 字节不是控制标志，而是数据的一部分，在熵编码部分，每一个 0xFF 字节后都插入 0x00 字节。

2. JPEG 编码器控制代码

编码过程由几个数据处理和传输操作组成。编码器必须在 JPEG 比特流中插入一些头数据（帧头，扫描头等），以方便解码。标准规定 JPEG 文件必须包含所有解码必需的表格，因此编码器必须完成一些辅助的与传输相关的功能以实现图像压缩。

控制函数 jpgenc_ti()在文件‘jpgenc_ti.c’中，它按时间顺序调用所有编码器部分的例程，如 DCT、量化、行程编码、变长编码等，并且完成数据在编码器程序间的传输。直接存储器传输需要两倍的缓冲空间，一方面用来从外部存储器读入图像数据，另一方面向外部存储器写出 JPEG 比特流，它控制在编码过程中屏蔽数据传输。

图 3.10 为编码器控制流程的示意图。

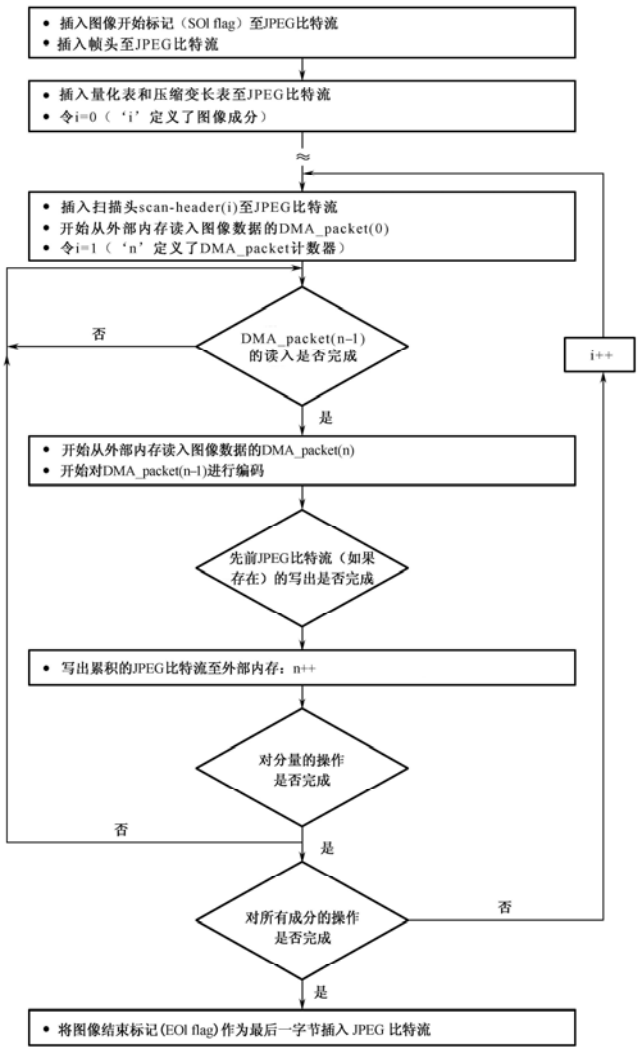


图 3.10　编码器控制流程

函数 jpegenc_ti() 包括全部编码器运行程序的全部控制逻辑，它通过驱动函数调用，驱动函数需要必要的参数。驱动原型参见‘main_interface.c’文件。函数 jpegenc_ti()的格式如

下所示：

```
jpegenc_ti (const unsigned char       sample_prec,
       const unsigned char         num_comps,
       const unsigned char         num_qtables,
       const unsigned char         interleaved,
       const unsigned int      format,
       const unsigned short        *num_lines,
       const unsigned short        *num_samples,
       unsigned char           **raw_img,
       unsigned char           *output_area);
sample_prec: 8-12 bit image samples (only 8-bits currently supported)
num_comps:    3 - color; 1 - grayscale
num_qtables: 2 - color; 1 - grayscale
interleaved: 1 - interleaved; 2 - non-interleaved
format:   0x01110111 - 4:4:4; 0x01120112 - 4:2:0; 0x01110112 - 4:2:2
num_lines:  Vertical dimension of the frame in terms of lines
num_samples:  Horizontal dimension of the frame in terms of samples
raw_img:  Pointer to the three memory areas that hold the image data
```

3. JPEG 编码器辅助函数

（1）帧头格式说明：帧头格式参数功能在函数 framehdr_spec( )中完成。这段程序在比特流中插入标准格式报头。格式报头包括将传送至解码器的图像参数，如图像的高度、宽度像素等。

（2）量化说明：量化表在函数 quant_table_spec( )中规定，这段程序将量化表格插入 JPEG 比特流中。

（3）变长编码说明：变长编码在程序 Huffman_tables_spec( )中实现，这段程序在 JPEG 比特流中插入哈夫曼压缩的比特流。

（4）扫描报头说明：扫描头在程序 scanhdr_spec( )中实现，它在比特流中插入标准的扫描头。这一扫描头包括解码器所需的图像分量的参数，如该扫描中图像分量数，解码所需的表格等。

4. JPEG 编码器应用编程接口（API）

应用编程接口的封装源自 TMS320 DSP 算法标准文档。了解算法标准是理解应用编程接口的关键。完整地讨论如何构成 eXpressDSP 算法已经超出本节的范围，但将讨论确保算法可以运行的算法接口。如果一个算法实现 IALG 接口并遵守算法标准的编程规则，那么它就是 eXpressDSP 兼容的。ALG 接口的核心是 IALG_Fxns 结构类型，其中定义了几个函数指针。每个 eXpressDSP 兼容的算法必须定义并初始化一个 IALG_Fxns 类型的变量。在 IALG_fxns 中，必需的函数有 algAlloc()，algInit()和 algFree()，其余函数是可选择的。

```
typedef struct IALG_Fxns {
Void   *implementationId;
```

```
        Void   (*algActivate)(IALG_Handle);
        Int        (*algAlloc)(const IALG_Params *, struct IALG_Fxns **, IALG_
MemRec *);
        Int        (*algControl)(IALG_Handle, IALG_Cmd, IALG_Status *);
        Void   (*algDeactivate)(IALG_Handle);
        Int        (*algFree)(IALG_Handle, IALG_MemRec *);
        Int        (*algInit)(IALG_Handle,const IALG_MemRec *,IALG_Handle,const
        IALG_Params *);
        Void   (*algMoved)(IALG_Handle,const IALG_MemRec *,IALG_Handle,const
        IALG_Params *);
        Int        (*algNumAlloc)(Void);
        } IALG_Fxns;
```

这一算法实现了 algAlloc()函数，通过填充 memTab 结构告知结构框架它的内存需求。同时它也告知结构框架是否存在此算法的父对象。基于调用 algAlloc()得到的信息，结构框架分配内存。algInit()函数初始化 algAlloc()函数中申请的固定内存。经过调用函数 algInit()后，句柄指向的算法场景就可以使用了。结束使用算法内存用 algFree()函数。算法应该释放掉每个用 algAlloc()申请的内存的地址和大小，以便删除对象而防止内存漏洞。

JPEG 编码器的 API 如下：

```
        /*
        * ======== ijpegenc.h ========
        * IJPEGENC Interface Header
        */
        #ifndef    IJPEGENC_
        #define    IJPEGENC_

        #include <std.h>
        #include <xdas.h>
        #include <ialg.h>
        #include <ijpeg.h>
        /*
        * ======== IJPEGENC_Handle ========
        * This handle is used to reference all JPEGENC instance objects
        */
        typedef struct IJPEGENC_Obj       *IJPEGENC_Handle;
        /*
        * ======== IJPEGENC_Obj ========
        * This structure must be the first field of all JPEGENC instance objects
        */
        typedef struct IJPEGENC_Obj {
        struct IJPEGENC_Fxns   *fxns;
        } IJPEGENC_Obj;
```

```
        /*
        * ======== IJPEGENC_Params ========
        * This structure defines the creation parameters for all JPEGENC objects
        */
        typedef struct IJPEGENC_Params {
           Int size; /* must be first field of all params structures */
           unsigned int sample_prec;
           unsigned int num_comps;
           unsigned int num_qtables;
           unsigned int interleaved;
           unsigned int format;
           unsigned int quality;
           unsigned int num_lines[3];
           unsigned int num_samples[3];
           unsigned int output_size;
        } IJPEGENC_Params;
        typedef IJPEGENC_Params IJPEGENC_Status;
        /*
        * ======== IJPEGENC_PARAMS ========
        * Default parameter values for JPEGENC instance objects
        */
        extern IJPEGENC_Params IJPEGENC_PARAMS;
        /*
        * ======== IJPEGENC_Fxns ========
        * This structure defines all of the operations on JPEGENC objects
        */
        typedef struct IJPEGENC_Fxns {
           IALG_Fxns ialg; /* IJPEGENC extends IALG */
           XDAS_Bool (*control)(IJPEGENC_Handle handle, IJPEG_Cmd cmd, IJPEGENC_
Status
                     *status);
           XDAS_Int32 (*encode)(IJPEGENC_Handle handle, XDAS_Int8* in, XDAS_Int8*
out);
        } IJPEGENC_Fxns;
        #endif /* IJPEGENC_ */
```

5. JPEG 编码器性能

JPEG 编码器性能已经被广泛地用视频信号源进行了测量，表 3.5 是基于 C6201 EVM 和 C6211 DSK 进行的性能测量结果。

表 3.5　JPEG 编码器性能

图像分辨率	f/s With 200 MHz C6201	f/s With 150 MHz C6211
128×128（4∶2∶0）	569 f/s	382 f/s
256×256（4∶2∶0）	156 f/s	106 f/s
352×288（4∶2∶0）[CIF resolution]	104 f/s	69 f/s
640×480（4∶2∶0）[VGA resolution]	36 f/s	24 f/s
720×480（4∶2∶0）[SDTV resolution]	32 f/s	21 f/s

C6211 performance data based on [48K cache/16K SRAM] configuration. Recommended for JPEG.

3.2.2　JPEG 解码器

1. JPEG 解码器算法

图 3.11 为 JPEG 解码器处理过程的总体流程。

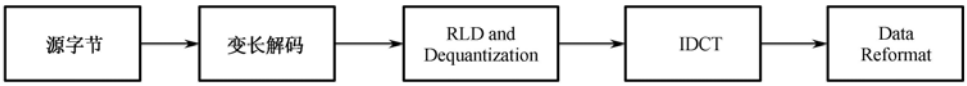


图 3.11　JPEG 解码器

（1）源字节：在 JPEG 标准中，控制标志字为 0xFF。这一标志后面接一个或多个控制指令。在 0xFF 字节后面如果接 0x00 字节，就表示 0xFF 字节不是控制标志字，而是数据的一部分。

（2）变长解码 (VLD)**：**变长解码将 JPEG 比特流解码并在 DCT 域产生图像数据。解码分为两步：一是直流系数解码，二是交流系数解码。解码过程实现了快速解码。

（3）游程解码与反量化：这一步骤通过变长解码程序将量化后的直流系数进行解码，在 Zig-Zag 型扫描中计算和解码 AC 系数。类似于在计算直流系数的情况，反量化所有的非零系数。

（4）IDCT：这一操作对 8×8 数据块进行 2D 反离散余弦变换，输出相应 8×8 图像数据样本。

$$S_{yx}=\frac{1}{4}\sum_{u=0}^{7}\sum_{v=0}^{7}C_uC_vS_{vu}\cos\frac{(2x+1)u\pi}{16}\cos\frac{(2x+1)v\pi}{16}$$

其中，当 u, v=0 时，C_u，C_v＝$1/\sqrt{2}$；否则，C_u，C_v＝1，S_{vu} 是在 u,v 处的 DCT 成分，S_{yx} 是图像像素在 x, y 处的空间样本值。

如下所述，将 2D 反离散余弦变换分成两个 1D 操作，以减少操作处理的数量：

- 完成 8 个 1D IDCT，每一个都对应排列中的行。
- 完成 8 个 1D IDCT，每一个都对应排列中的列。

（5）数据重定格式：数据重新定格式，例程将块图像数据转化成按光栅扫描的图像数据结构。图 3.12 为重新定格前的解码图像数据。

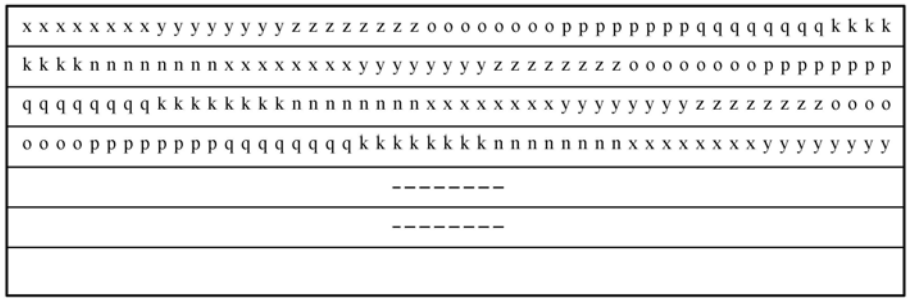


图 3.12　重新定格式前的解码图像数据

重新定格式后的图像格式如图 3.13 所示，并将动态范围从−128～127 调整到 0～255，以便在显示器输出。

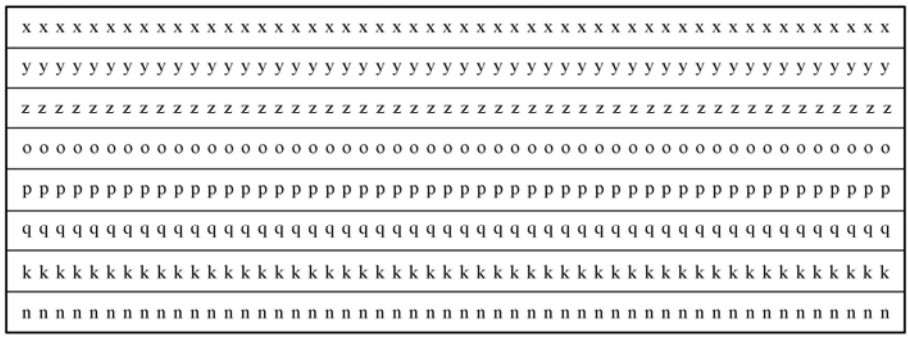


图 3.13　光栅扫描重新定格式的图像数据

2. JPEG 解码器控制代码

控制函数 jpgdec_ti()在文件‘jpgdec_ti.c’中，它按次序调用所有的解码器例程，如变长解码、游程解码、反量化、IDCT 等，并且完成数据在解码器例程间的传输。直接存储器控制需要两倍的缓冲空间，一方面用来从外部存储器读入压缩数据流，另一方面向外部存储器写出 JPEG 解码的图像数据，它控制在解码过程中屏蔽数据传输。

函数 jpegdec_ti()包括解码器应用的全部控制逻辑，它由驱动函数调用，驱动函数带有必要的参数。驱动函数原型参见‘main_interface.c’文件。函数 jpegdec_ti()的格式如下所述：

```
jpegdec_ti (unsigned char *jpg_bit_stream,
            unsigned char *output_addr );
jpg_bit_stream: The pointer to the JPEG bit-stream (input)
ouput_addr: The pointer to the output data (Y-Cr-Cb)
```

图 3.14 为解码器控制流程图。

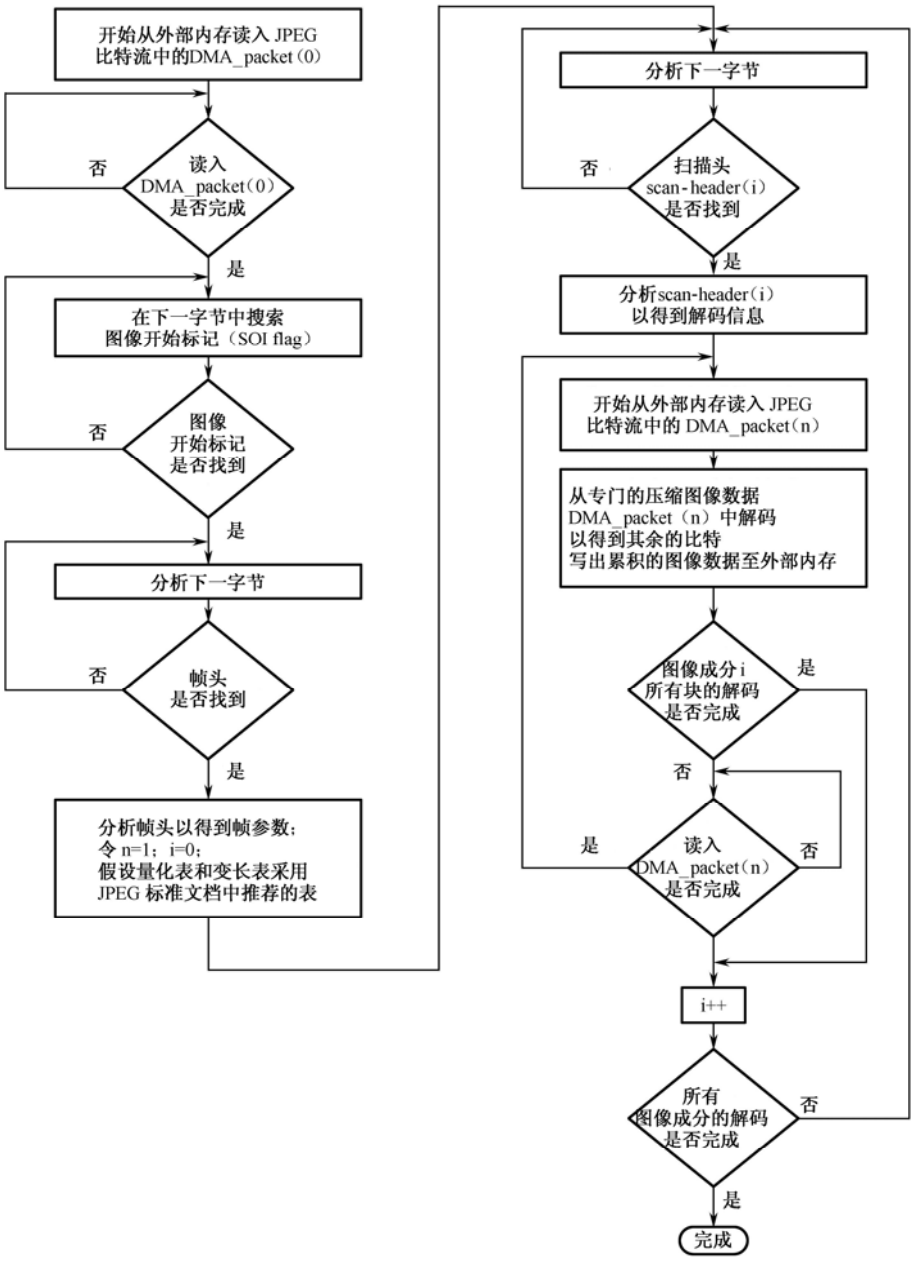


图 3.14　解码器控制流图

3. JPEG 解码器应用编程接口

API（应用编程接口）封装要遵从 TMS320 DSP 运算标准文档提供的模板。了解算法标准文档是理解应用编程接口的基础，这个内容超出本节范围。JPEG 解码器应用编程接口为：

```
/*
* ======== ijpegdec.h ========
```

```
* IJPEGDEC Interface Header
*/
#ifndef IJPEGDEC_
#define IJPEGDEC_

#include <xdas.h>
#include <ialg.h>
#include <ijpeg.h>
/*
* ======== IJPEGDEC_Handle ========
* This handle is used to reference all JPEG_DEC instance objects
*/
typedef struct IJPEGDEC_Obj *IJPEGDEC_Handle;
/*
* ======== IJPEGDEC_Obj ========
* This structure must be the first field of all JPEG_DEC instance objects
*/
typedef struct IJPEGDEC_Obj {
   struct IJPEGDEC_Fxns *fxns;
} IJPEGDEC_Obj;
/*
* ======== IJPEGDEC_Params ========
* This structure defines the creation parameters for all JPEG_DEC objects
*/
typedef struct IJPEGDEC_Params {
   Int size; /* must be first field of all params structures */
} IJPEGDEC_Params;
/*
* ======== IJPEGDEC_Status ========
* This structure defines the status parameters for all JPEG_DEC objects
*/
typedef struct IJPEGDEC_Status {
   Int size; /* must be first field of all params structures */
   unsigned int num_lines[3];
   unsigned int num_samples[3];
   unsigned int gray_FLAG;
   unsigned int outputSize;
} IJPEGDEC_Status;
/*
* ======== IJPEGDEC_PARAMS ========
* Default parameter values for JPEG_DEC instance objects
*/
extern IJPEGDEC_Params IJPEGDEC_PARAMS;
```

```
        /* ======== IJPEGDEC_Fxns ========
        * This structure defines all of the operations on JPEG_DEC objects
        */
        typedef struct IJPEGDEC_Fxns {
           IALG_Fxns ialg; /* IJPEGDEC extends IALG */
           XDAS_Bool  (*control)(IJPEGDEC_Handle  handle,  IJPEG_Cmd  cmd,
IJPEGDEC_Status *status);
           XDAS_Int32 (*decode)(IJPEGDEC_Handle handle, XDAS_Int8 *in,
                       XDAS_Int8 *out);
        } IJPEGDEC_Fxns;
        #endif /* IJPEGDEC_ */
```

4. JPEG 解码器性能

针对不同图像和压缩率广泛测量 JPEG 解码器性能，表 3.6 是基于 C6201 EVM 和 C6211 DSK 的性能测量结果。

表 3.6　JPEG 解码器性能

图像分辨率	f/s With 200 MHz C6201	f/s With 150 MHz C6211
128×128（4∶2∶0）	528 f/s	374 f/s
256×256（4∶2∶0）	159 f/s	108 f/s
352×288（4∶2∶0）[CIF resolution]	107 f/s	72 f/s
640×480（4∶2∶0）[VGA resolution]	39 f/s	26 f/s
720×480（4∶2∶0）[SDTV resolution]	35 f/s	23 f/s

C6211 performance data based on [48K cache/16K SRAM]configuration. Recommended for JPEG.

本章介绍了 JPEG 标准的基本原理，并介绍了在 TMS320C6000 平台上实现 JPEG 基本编码器的流程和性能。

参 考 文 献

［1］A. Narayan. J. Min, V. Markandey. TMS320C6000 JPEG implementation. TI Application Report, SPRA704.

［2］张旭东，卢国栋，冯健．图像编码基础和小波压缩技术．北京：清华大学出版社，2004.

注：3.2 节叙述的JPEG编码器的原代码可通过 www.ti.com网站下载。

第 4 章　MPEG 编码标准及其在 DSP 上的实现

MPEG 系列标准是国际标准化委员会 ISO 制定的一系列多媒体编码表示的标准，目前公布的有 MPEG-1、MPEG-2、MPEG-4、MPEG-7 和 MPEG-21，与 DSP 应用相关的主要是前三个标准。

MPEG-1 标准是为了音视频存储媒体（如 VCD）的应用。该标准的视频编码部分在典型的运动补偿预测编码（MCPC）框架的基础上，应用了半像素的双向预测技术，提供更好的编码质量和更高的压缩比，其主要应用目标为 1～1.5 Mb/s 的情况下，提供 25 帧 CIF（352×288）VHS 质量的图像。

MPEG-2 保留了 MPEG-1 的基本技术，并针对分场的图像格式，使用了帧、场联合的编码结构，能够更好地支持高分辨率的图像和高质量视频编码。目标码率在 3～35 Mb/s 的传输速率条件下，提供目前广播级 TV 到 HDTV 的图像编码，同时，能够提供质量、时间、空间等多种可伸缩性的（分级）编码模式。MPEG-2 不像 MPEG-1 那样只是用于静态存储媒体，而是把应用目标扩展到单向通道的视频广播领域，比如卫星数字电视和有线电视信号。

MPEG-4 则是针对更加广泛的应用，定义了更加灵活的编码器结构。

本章只讨论 MPEG 标准中视频编码部分，首先概要介绍 MPEG 视频编码标准的基本技术，然后介绍一个在 TI DSP 上实现 MPEG-2 解码的例子。

4.1　MPEG-1 视频压缩标准

MPEG-1 是 MPEG 第一阶段的成果，它的编号为 ISO/IEC 11172，它规定视频信息与伴音信息经压缩之后的数据速率上限为 1.5 Mb/s，从而可以在 CD-ROM、硬盘、可写光盘、数字音频磁带（DAT）等介质上进行存储，也可以在局域网、ISDN 上进行视频与伴音信息的传输。

ISO/IEC 11172 在总标题“信息技术——上限为 1.5 Mb/s 的数字存储媒体上的运动图像及伴音的编码”之下，包含下列四部分。

第一部分：系统，这部分主要论述了如何将符合该标准的视频和音频部分的一条或多条数据流与定时信息相结合，形成单一的复合流。

第二部分：视频，这部分提供了一种统一的编码格式，用来描述存储在各种数字存储媒体里的经压缩的视频信息。

第三部分：音频，这部分阐述供媒体存储的高质量音频的编码表示和高质量音频信号的解码方法。

第四部分：一致性测试。

本节只讨论 MPEG 视频压缩部分。

MPEG-1 视频编码算法是一种有损压缩算法，它适用于多种视频输入格式并且应用范围很广。应注意的是，它特别对比特率为 1.5 Mb/s 的连续传输和存储应用（比如 CD-ROM）进行了优化。

MPEG-1 特地采用术语图片而不是帧表示画面，这是因为它不支持隔行扫描所得的视频源数据流。在隔行扫描中，一帧图像由两场（Field）组成，即所谓的奇场和偶场。在一帧画面中，两场的扫描线是交织在一起的，所以，空间上相邻的行在时间上是不相邻的。比如说帧率为 25 f/s，则空间上相邻的两行在时间上相差 1/50 s。与隔行扫描相对应的是逐行扫描，在逐行扫描中没有场的概念，空间上相邻的行在时间上也是相邻的。扫描行从左上角开始连续扫描整幅画面。因此，在 MPEG-1 中，由电视信号所产生的隔行扫描的视频流在被编码前必须要转化为逐行扫描的结构, 这个过程不在 MPEG-1 的标准中，因此 MPEG-1 只是将输入图像当做按时间排列的图片序列，而不管它的来源是否是隔行还是逐行扫描，尽管如此，我们还是用帧内和帧间这样的词形容所采用的编码技术。

MPEG-1 标准在制定时迎合了两种关键的需求，即高压缩比和对编码比特流的随机存取。单独的帧内编码十分适用于要求随机存取的场合，但是却不能满足高压缩比的要求。为了同时满足这两种相互矛盾的需求，MPEG 采用了帧内编码和帧间编码相互结合的折中策略。

MPEG 中的压缩功能主要包括：

（1）对源视频码流中的亮度和色彩分量进行时间和空间上的亚采样，以降低数据源的码率。

（2）基于宏块（MB）的运动补偿。

（3）基于块的 DCT 变换。

（4）对运动矢量和量化 DCT 系数进行 Huffman 压缩编码。

举个例子，一个视频流中，画面大小为 352×240，每像素 12 b，每秒 30 幅画面，MPEG 可以把这个 30.4 Mb/s 的码流压缩为 1.15 Mb/s。尽管压缩率达 26∶1，画面质量仍然可以和模拟录像机的质量相媲美。

4.1.1　SIF 格式

MPEG-1 的比特流语法允许输入画面的最大尺寸达 4 095×4 095，但是大多数的 MPEG-1 应用特别对 SIF 输入格式进行了优化，SIF 格式是 ITU－R BT.601 的一种简单变体。根据 ITU－R BT.601，一个彩色视频源包含三个分量：一个亮度信号（Y）和两个色差信号(Cr，Cb)。ITU－R BT.601 有两种分辨率选择，一种是适用于 NTSC 电视系统的 525 行/帧，亮度信号的分辨率为 720×480，每个色差信号的分辨率为 360×480，这就是通常说的 4∶2∶2 亚采样格式。第二个选项适用于 PAL 电视系统，625 行/帧，25 f/s，这里，亮度信号的分辨率为 720×576，而色差信号的分辨率为 360×576。为了使压缩后的码率降到 1.5 Mb/s 以下，MPEG 定义了 SIF（Source Input Format）格式。SIF 序列的亮度信号分辨率为 360×240，每秒 30 幅画面或者是 360×288，每秒 25 幅画面。在两种情况中，色差信号的分辨率在水平和垂直方向上都是亮度信号的一半。这就是所谓的 4∶2∶0 亚采样格式。SIF 可以很容易由对 ITU－R BT.601 进行滤波和亚采样得到。

在 MPEG-1 中，三种彩色信号总是组合在一起。一个宏块（Macro Block）被定义为包含

4 个 8×8 的亮度块、1 个 8×8 Cb 色度块和一个 8×8 Cr 色度块。因此，一个宏块的最大维数是 16×16 像素。每幅画面均被由左到右，由上到下分割为一系列的宏块。这也要求原始图像的水平和垂直分辨率都是 16 的倍数。如果不满足这一点，编码器会在每幅画面的右侧和底部加入填充像素，这些像素以后会被解码器所丢弃。因为 SIF 格式画面的水平分辨率为 360 不是 16 的整倍数，我们可以把每行的最后一个像素复制 8 次，这样每行就有了 368 像素。另一种可行的办法是，丢弃每行最左的 4 像素和最右的 4 像素，这样剩下的画面每行包含 352 像素，我们称这种画面为重要像素区。表 4.1 总结了 ITU－R BT.601，SIF 和重要像素区的大小，所以，实际情况下，经常将 SIF 分辨率说成是 352×288。

表 4.1　ITU－R BT.601，SIF 和重要像素区的画面大小

每秒画面数	30	25
CCIR 601		
Y	720×480	720×576
Cb，Cr	360×480	360×576
SIF		
Y	360×240	360×288
Cb，Cr	180×120	180×144
SIF 重要像素区		
Y	352×240	352×288
Cb，Cr	176×120	176×144

MPEG-1 提供了很大程度上的灵活性，但是要求每个 MPEG 解码器支持所有的编码选项是不现实的。因此，MPEG-1 定义了受限参数比特流，所有的 MPEG-1 兼容解码器都要支持具有该特性的比特流。从表 4.2 中可见，像素传输率 396×25 宏块/秒限制了画面的大小，近似应为 352×288，即 25 Hz SIF 图像的重要像素区的大小。

表 4.2　受限参数比特流的编码限制

编 码 参 数	最 大 值
图像水平大小	768 像素
图像垂直大小	576 像素
宏块数目	396
像素速率	396×25 宏块/s
图片速率	30 图片/s
运动矢量范围	+/−64 像素(半像素精度)
输入缓存大小	327 680 b
比特率	1 856 kb/s

4.1.2　MPEG-1 视频编码

1. 画面类型

在视频序列上经常进行的一般操作包括视频编辑、随机存取和搜索查询，支持这些常用

的操作往往和高压缩比的要求是相矛盾的。MPEG-1 定义了 4 种类型的画面以达到两者的折中。

帧内编码帧（Intra-Pictures，以下简称 I 帧）应用帧内编码的方法实现压缩，它们编码时并没有参考视频序列中的其他帧。I 帧支持随机存取，但是压缩比低，这种编码方式和 JPEG 编码方式十分相似。预测编码帧（Predicted Pictures，以下简称 P 帧），编码时要利用过去的 I 帧或 P 帧进行运动补偿预测，P 帧的压缩比要高于 I 帧。双向预测帧（Bidirectionally Predicted Pictures，以下简称 B 帧）提供了最高的压缩比，它们在运动补偿预测编码时，用到了“过去”的和“将来”的 I 帧或 P 帧。B 帧不被用于其他 B 帧或 P 帧的运动补偿预测，可以容忍更大的失真并提供相对于 I 帧和 P 帧更大的压缩比。DC 编码帧（DC Coded Pictures，以下简称 D 帧）类似于 I 帧，但是只有 DCT 变换结果的 DC 项系数被保留。D 帧不能用于对其他帧的预测中，定义它的目的是提供一种快速搜索的方法，因此 D 帧一般不常用。

图 4.1 显示了在一个 8 画面的视频序列中，3 种主要类型帧的关系。

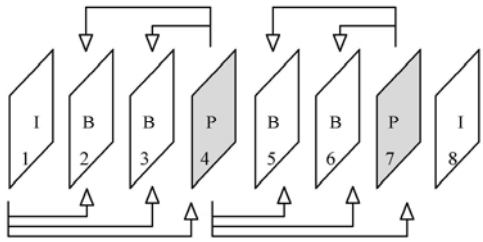


图 4.1　3 种主要帧类型的相互关系实例

画面 p1 和 p8 为 I 帧，p4 和 p7 为 P 帧，剩下的为 B 帧。在这个例子中，p2，p3 均利用 p1 和 p4 进行运动补偿预测编码。MPEG 中对于 P 帧和 B 帧的使用并没有任何要求。一个对典型测试序列的实验结果表示：对 SIF 分辨率，采用 IPBBPBBPBBPBBPBB 的 GOP 结构，码流速率为 1.15 Mb/s 的 MPEG 视频序列中 I 帧，P 帧和 B 帧的平均每图像码率大小分别为：156 kb/s，62 kb/s 和 15 kb/s。可以注意到，B 帧的大小远远小于 I 帧和 P 帧，但是单纯增加 I 帧和 P 帧之间 B 帧的数量并不能获得更好的压缩比，这是因为这样做会导致 B 帧与相应的 I 帧和 P 帧的时间距离增大，时间相关性降低，降低了运动补偿预测的性能。

2. 编码过程

MPEG 标准并没有定义特定的编码过程，它只是定义了编码比特流的语法和解码过程。基于以上的要求，可以通过图 4.2 来表示一个 MPEG 编码器要完成的功能。

预处理：编码过程通常是由预处理开始的，可能的工作有，RGB 到 YC_bC_r 的色彩空间变换，格式转换（隔行到逐行的变换），预滤波和亚采样，等等，这些操作并没有在 MPEG-1 标准中给出。

运动估计和补偿：进行了预处理以后，编码器要为输入画面选择合适的编码方式。I 帧不需要运动补偿和估计。每个宏块进行 DCT 变换，之后对 DCT 系数进行量化，再对结果进行可变长度编码，最后被存储在帧缓冲区里。在每个宏块内部，是分别对每个 8×8 的块进行的。在一个需要恒定比特率的应用中，还需要一个缓存校正器随时调整量化矩阵，以使编

码后码流的比特率保持相对恒定。对于可变比特率的应用，缓存校正器是不需要的。

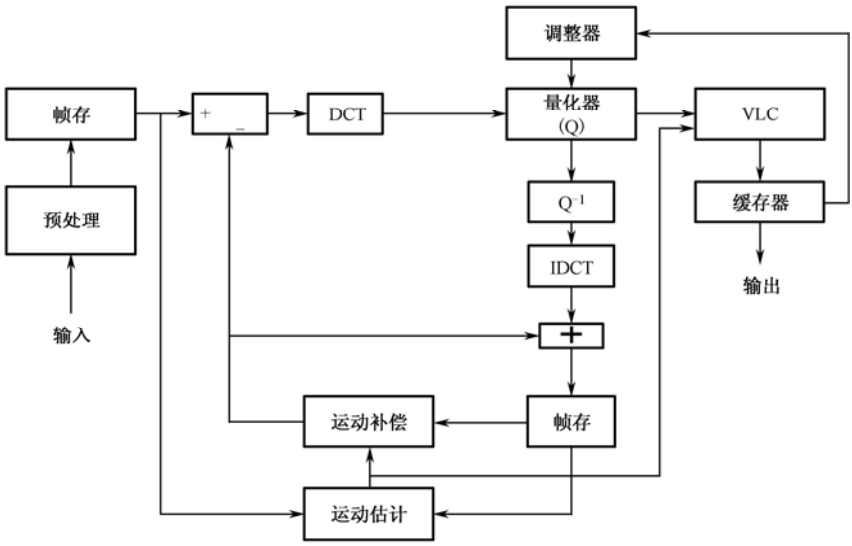


图 4.2　MPEG-1 编码器的典型结构图

在上述过程中，一个块在被量化后还要进行反量化（Q^{-1}）并进行逆 DCT 变换，重新变换回空域表示。这个操作和解码器的行为是同样的，它使得编码器获得了一个和在解码端进行解码后的图像相一致的副本。这个副本随即被存储到本地的存储器中，以便用于后续的预测编码。也就是说，后续编码时，使用的参考帧是已编码图像的解码重构而不是原图像。另外，这样做也可以使编码器可能监视被传输图像的质量使其不会与原始图像相差太多。之所以在量化之后就进行一系列逆变换或反变换，是因为 VLC 编码本身是无损的，因此，自然不需要把 VLC 单元包含在反馈回路之中。

如果输入的图像被编码为 B 帧或 P 帧，编码器并不直接对图像中的宏块进行编码而是对预测误差进行编码。在 MPEG 术语中，这被称为“帧间预测编码”，对于 P 帧，对当前图像中的每一个宏块，运动估计器在已编码的过去图像中的某一区域进行搜索，找到与这个宏块匹配的最好的一个宏块，并记下它的偏移坐标（x, y），这组坐标就是运动矢量。这两个宏块相减，并对差值进行 DCT 变换编码。关于基于宏块的运动估计算法在第 3 章已有详述，这里就不再赘述了。

对于 B 帧，运动估计过程要进行两次，一次对一个过去帧，一次对一个将来帧，所以 B 帧有两个运动矢量。编码器可以利用这两个候选宏块中的任意一个或这两者的平均值和当前宏块相减来得到预测误差。在 MPEG 的术语中，这被称为“帧间内插编码”。接下来对预测误差进行基于块的 DCT 变换。经过量化后，预测误差的 DCT 系数经过 Zig-Zag 扫描后和运动矢量一起再进行可变长度编码（VLC）。在 MPEG 中可变长度编码一般采用 Huffman 编码。

因为要用到双向预测，为了使解码器端能够正确解码，在编码端需要对输入的图像序列进行重排序。以图 4.1 中的图像序列为例，因为 p2 是一个 B 帧，它在运动补偿估计时用到了 p1 和 p4，所以它只能在 p4 输入之后才能进行编码。这样，在进行了输入重排序之后，正确的编码序列应为 p1，p4，p2，p3，p7，p5，p6，p8。

解码器在解码收到的每帧已编码图像时，就有足够的信息来进行正确的解码。当然，解码器还要对解码所得到的图像进行重排序，使得它们被按照正确的时序进行显示，这通常可以通过使用缓冲器实现。

3. 编码比特流的结构

MPEG-1 的语法定义的编码视频比特流分为 6 层：

序列层（Sequence Layer）；

图组层（Group of Pictures Layer）；

图像层（Pictures Layer）；

片层（Slice Layer）；

宏块层（Macroblock Layer）；

块层（Block Layer）。

序列层是编码流的最高层，它包含序列标题和后续的一系列图像组（GOP）。序列层始于序列开始码（0x000001b7，Sequence-Start-Code），结束于序列结束码（0x000001b3，Sequence-End-Code）。在序列标题中包含有一系列关于序列的信息，包括每幅图像的垂直和水平尺寸，像素宽高比，画面速率（画面/s，比特率（以 400 b/s 为单位），解码器所需的最小缓冲区大小（以 2 KB 为单位）。标题中还包含一位标志指出编码流是否是前面提到过的受限参数比特流。另外，标题中还可能包括帧间或帧内编码时用到的 DCT 系数量化矩阵以及一些用户定义的信息。默认的帧内编码系数量化矩阵是 JPEG 标准中给出的对亮度元素所使用的矩阵。对于非帧内编码帧，默认量化矩阵中所有的元素都为 16。之所以对非帧内编码帧的量化使用这种均匀量化的方式，是因为这种帧提供的是运动预测的误差信息，对各个频率的元素应该一视同仁。而对于帧内编码帧，由于各个频率元素的值和图像块的频率分布是直接联系的，因此，以采用非均匀量化忽略一些次要信息（通常是高频信号）。

一个图像组是一系列要连续编码的图像，一个图像组中至少含有一个 I 帧。一个图像组可以以 I 帧或 B 帧开始，而结尾必须为 I 帧或 P 帧。如果一个图像组以 I 帧或一个不依赖于以前图像组中图像的 B 帧开始，那么这个图像组就是独立的，称为“封闭图像组”（Closed GOP）。图像组层的标题中含有的信息包括：定时信息，封闭标志，一些扩展信息和用户定义信息。图 4.1 中的 p1～p7 就是一个包含 7 幅图像的图像组，而且是一个封闭图像组。

图像层定义了每幅图像的编码信息。由于可能要进行 P 帧和 B 帧的重排序，在图像层的标题中定义了一个时间参考数，用于定义各帧的显示顺序。其他的标题信息包括图像类型、同步、分辨率和运动矢量范围。

每幅图像都被切割成一系列的片。片的大小是可变的，最大可以是整幅图像，最小可以是一个宏块。定义片的目的主要是在发生误码时，解码器可以简单地丢弃一个片而不是整幅图像。片标题中的信息记录了片在图像中的位置以及该片的量化因子 Q，Q 在 1～31 之间，解码器在反量化时要用到它。定义量化因子可以使得编码器在片这个层次上对码流进行校正。

一个片被进一步划分为若干宏块。每个宏块的标题定义了宏块的类型，位置信息，水平和垂直运动矢量，以及在一个宏块中的那些块真正被编码和传送了（一个宏块中包含 6 个块，

一个 C_b 块，一个 C_r 块，4 个亮度块）。

图 4.3 显示了 MPEG-1 比特流中的语法结构。

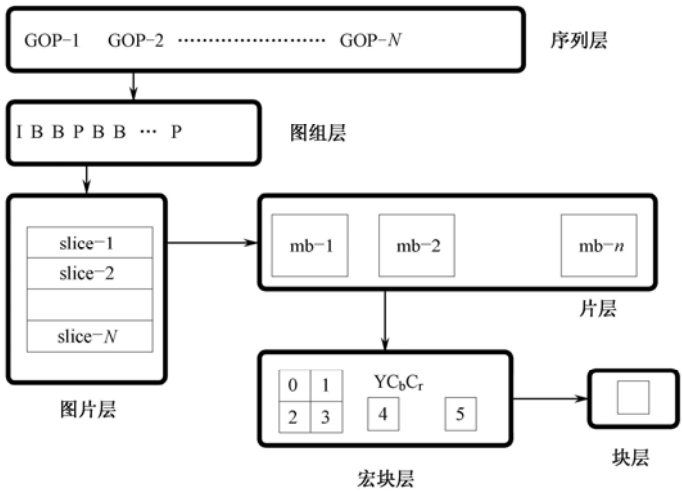


图 4.3　MPEG-1 比特流中的语法结构

4. 宏块的编码

前面提到过，MPEG-1 中定义了三种主要的图像类型：I 帧，P 帧和 B 帧，但是即使在单独的一个帧中，不同宏块的编码方式也可以是不同的，下面分情况讨论。

➢ **I 帧的编码**

I 帧编码时不进行运动估计，但是 MPEG 的语法结构允许每个宏块在编码时采用不同的量化矩阵。对块进行 DCT 变换之后，每个 DCT 系数都要与帧内编码量化矩阵的相应元素相除来进行量化。对于 DC 系数，量化步长通常固定为 8。对于 AC 系数，量化步长由量化矩阵中的相应元素和量化因子（在片的标题中定义）决定，一个默认的 I 帧量化矩阵是：

8	16	19	22	26	27	29	34
16	16	22	24	27	29	34	37
19	22	26	27	29	34	34	38
22	22	26	27	29	34	37	40
22	26	27	29	32	35	40	48
26	27	29	32	35	40	48	58
26	27	29	34	38	46	56	69
27	29	35	38	46	56	69	83

给定 DCT 变换结果 Dct（i, j），量化矩阵 Q（i, j），以及量化因子 q，量化后的 AC 系数 QDct（i, j）可以表示为:

$$\mathrm{QDct}(i, j) = \frac{8\mathrm{Dct}(i, j)}{qQ(i, j)}$$

每个经过量化的系数都被截取在范围[−255,255]中。量化因子 q 是定义在片的标题中的，

但是也可以在每个宏块的标题中重新定义这一参数。在 MPEG 的术语中，这个量化因子被定义为 MQUANT。I 帧中有两种类型的宏块，即使用新 MQUANT 的宏块和没有变化 MQUANT 的宏块。

量化之后，DC 系数与前一块的量化 DC 系数相减，对差值编码为（size,amp）对。amp 为差值的大小，若是正值则为其二进制表示，若是负值则采用反码表示，size 表示 amp 所需的二进制位数，这与 JPEG 类似。举个例子，量化 DC 系数的差为 195，则：

```
size=8 (195=0b11000011)；amp=11000011;
```

若差值为-195，则：

```
size=8，   amp=00111100；
```

对 size 进行 Huffman 编码，amp 的值跟在 Huffman 码之后，比如，如果对应于 size=8 的 Huffman 码为 111110，则上述的 DC 系数差为-195，在编码码流中的最终表示为：

```
11111000111100
```

解码时先对 111110 解码为 8，再从后取得 8 位二进制数，即可还原出直流系数差值。

对于 AC 系数，编码器首先对它们按照 Zig-Zag 顺序进行排序，Zig-Zag 排序本质上是按照 DCT 输出中元素代表的频率分量，由低频到高频排序，这样有利于高效的压缩（因为高频分量一般都是次要信号，而且幅值一般较小）。之后每个非零的 AC 系数被表示为行程/幅度对，行程表示在该非零 AC 系数之前有多少个零值系数，而幅度则表示了该系数的值。MPEG 中定义了常用的行程/幅度对的 Huffman 码表，对其进行 Huffman 编码，而表中未定义的行程/幅度对则编码为 escape 码，后跟它们单独的码字。有关各种码表的详细列表，参考 MPEG-1 文档。

- **P 帧的编码**

P 帧和 I 帧一样被划分成片和宏块，由于运动补偿的原因，MPEG-1 编码器对 P 帧中的宏块进行编码时有更多的选择，决定如何对一个 P 帧中的宏块编码，可按下述步骤执行：

（1）决定是否使用运动补偿（即是否把运动矢量置零）。在许多情况下，使用非零的运动矢量所形成的预测误差并不比使用零值的运动矢量所形成的预测误差小多少，考虑到非零的运动矢量需要额外的编码比特，因此，使用零值的运动矢量（或者说不采用运动补偿），在一些情况下（如背景区）有利于提高编码效率。

（2）决定对宏块使用帧间编码方式还是帧内编码方式。在许多情况下，对某些宏块采用帧内编码的方式也许会需要更少的比特，这通常发生在由于运动十分剧烈（时间活跃度很高）而导致运动估计失败的时候。

（3）决定宏块是否要编码。有时，在量化之后，宏块中所有的 DCT 系数都是零，这种宏块就不需要编码，称它为被跳过的宏块。在对这种宏块解码时，只需要从过去帧中把对应的宏块（由运动矢量决定）复制到这个宏块中就行了。更进一步，如果一个宏块被编码了，也不是其中的所有 8×8 块都要被编码，因为可能在量化之后，某一个块或多个块的 DCT 系数都是零。在宏块的标题中定义了一个 6 位的块编码图案，用来指示宏块中的哪几个块被编码了。要注意的是，I 帧中没有被跳过宏块的说法，所有的块都必须被编码。

（4）决定是否需要改变 MQUANT。编码器为了使缓存不出现上溢或下溢的情况，可以对宏块的量化尺度因子单独调节。

➢ **B 帧的编码**

B 帧中宏块的编码方式和 P 帧中类似。

（1）决定是使用前向运动补偿，还是后向运动补偿或者是内插运动补偿。

（2）决定采用帧内编码的方式还是帧间编码的方式。

（3）决定宏块是否可以被跳过。

（4）决定量化尺度因子是否可以被改变。

表 4.3 给出了一个包含 150 幅画面的 MPEG-1 码流 I，P 和 B 帧的分布的例子。在这个码流中，每个图像组包含 15 幅画面并且每隔两个 B 帧有一个 P 帧，码流比特率是 1.15 Mb/s。

表 4.3　不同宏块编码类型在一个样本码流中的分布

画面类型	宏块类型				
	I	P	B	零运动矢量	跳过块
I	3 300				
P	897	8 587		5 128	568
B	60	7 356	22 845		429

注意，在 B 帧中有相当多的宏块是预测编码宏块（P-macroblock），这经常出现在有场景切换时或者出现在某一个B帧之前的P帧中存在的景物在该B帧之后的P帧中消失的情况下。

4.1.3　MPEG-1 视频解码

图 4.4 显示了一个 MPEG-1 视频解码器的方框图，可以看到，它和 4.1.2 节中提到的编码器框图中的反馈环十分相似。对输入的编码图像流首先要分析码流，确定它的类型（I,P,B），然后进行 Huffman 解码，之后对每一个宏块进行反量化并利用逆 DCT 变换使其由频域表示恢复到空域表示。以编码序列 p1，p4，p2，p3，p7，p5，p6，p8 为例，可以描述以下整个的解码过程。为简单，假设 P 帧中的所有宏块都采用预测编码，B 帧中的所有宏块都采用双向预测编码，这些条件并不是必需的，MPEG-1 对于宏块的编码具有很强的灵活性。下面来看一看示例序列的解码过程。

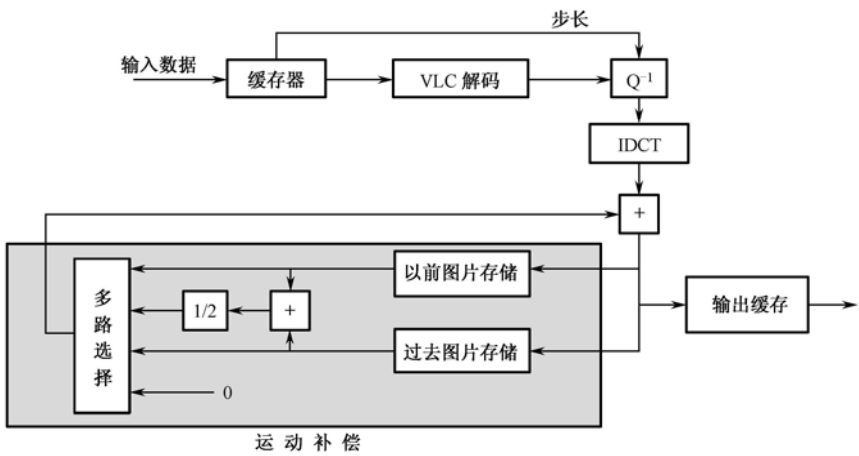


图 4.4　MPEG-1 视频解码器方框图

（1）输入图像 p1（I 帧），由于没有进行运动补偿，所以直接进行逆 DCT 变换，然后把解出的图像进行显示并存入过去帧缓存器中，（图 4.4 中的运动补偿多路选择器输入选择 0）。

（2）输入图像 p4（P 帧），对每一个宏块进行逆 DCT 变换并进行运动补偿，运动补偿意味着把在过去帧（这里为 p1）中由运动矢量指出的宏块与逆 DCT 变换的结果相加。重建的图像放入将来帧缓存器中。

（3）输入图像 p2（B 帧），进行逆 DCT 变换后进行双向的运动补偿，也就是利用 p2 的两个运动矢量所指出的过去帧（p1）和将来帧（p4）中的相应宏块形成对 p2 的预测值，并把这个预测值与逆 DCT 变换的输出相加，得到重建的图像。由于 p2 在显示时序上是紧跟在 p1 后面的，因此此时可以对 p2 进行显示。p2 不用被保存在任何一个帧缓存器中，因为如前所述，B 帧不参与其他帧的运动估计。

（4）输入图像 p3（B 帧），重复 p2 的解码过程，解码之后立即显示 p3。

（5）输入图像 p7（P 帧），重复 p4 的解码过程，重建的图像要放入过去帧缓冲器中（覆盖 p1），显示图像 p4。

（6）输入图像 p5，p6（B 帧），重复 p2 的解码过程，使用 p4 和 p7 进行运动补偿，解码后的显示顺序为 p5，p6，p7。

这样就完成了一组（GOP）图像的解码操作。下一组图像从 p8 开始，重复上述的解码过程。由于每个时刻最多有三幅图像需要存储（过去帧，当前帧，将来帧），对于 SIF 格式的图像序列来说，MPEG-1 解码器需要最少 500 KB 的缓存区。

4.1.4　MPEG-1 的其他问题

复杂度估计显示，MPEG-1 解码器可以在通用计算机上用软件实现。表 4.4 显示了 MPEG-1 各解码功能的计算负载。这些数据是基于 SIF 格式分辨率的图像，比特率为 1.15 Mb/s，每两个 B 帧一个 P 帧运算负载是由实际统计获得。

一般说来 IDCT 是 MPEG-1 解码中耗时最多的一项操作，但有时也未必如此，如果编码器能够进行十分有效的运动估计和码率控制，如表 4.4 所示，那么许多块在解码时并不需要 IDCT，这时，运动补偿就成为最耗时的操作。另外，YC_bC_r 到 RGB 的色彩空间转换和插值的耗时也是比较大的。

表 4.4　MPEG-1 各解码功能的计算负载

解 码 功 能	负　　载/ %
标题解析	0.44
Huffman 解码和反量化	19.00
逆 DCT 变换	22.10
运动补偿	38.64
色彩空间转换和显示	19.82

4.2　MPEG−2

MPEG-2 是在 MPEG-1 的基础上进一步发展成的音视频编码标准，MPEG-2 的主要目标

是针对广播级的高质量音视频应用，尤其能够很好地处理隔行扫描的数字视频。MPEG-2 取得了很大的商业成功，SDTV、HDTV、DVD 和 DVB 等新的应用产品都已采用 MPEG-2 的不同层进行实现。MPEG-2 的制定工作起始于 1990 年，至 1994 年完成了它的前 3 部分的标准化工作，即系统、音频和视频。目前 MPEG-2 已增加到 10 部分，其中包括模拟软件、数字存储媒体的命令和控制（DSM－CC）以及高级音频编码（AAC）等。本节仅简要介绍 MPEG-2 视频部分相对 MPEG-1 增加的主要特性。

MPEG-2 的视频编码算法是在 MPEG-1 基础上发展的，对 MPEG-1 有向下的兼容性，但又增加了许多新的特性，以下是 MPEG-2 和 MPEG-1 的主要区别。

（1）支持多种输入视频序列的采样格式，如 4：2：2 和 4：2：0。其中在 4：2：0 格式时，色度信号的采样位置与以前的 SIF 和 CIF 格式下色度信号采样位置相比，向左移动了半个像素。

（2）支持几种可选择的运动预测模式，如按帧或按场的运动预测，16×8 的运动预测和补偿等。

（3）按帧或按场的 DCT 和两种不同的扫描方式。

（4）支持几种分级的编码模式：数据划分、空间分级、时间分级和 SNR 分级。空间分级是在同一个码流中可以解码出不同的图像尺寸，或者说是不同的空间分辨率；时间分级是指同一个码流可以解码出不同的帧率；SNR 分级是指由同一个码流可以解码出相当于按不同量化步长产生的不同级别的图像质量。

（5）提供不同的档（Profile）和层（Level）的组合，它们提供 MPEG-2 的不同实现工具和不同参数取值范围，档和层的指示字放在码流的头文件中，指示解码器，使解码器判断能否具有解码此码流的能力。

（6）非线性量化表和对 DCT 系数新的 VLC 码表。

以下简要介绍其中的一些新特性。

4.2.1　MPEG-2 的运动估计

MPEG-2 针对隔行视频设计了各种可选的运动预测方法。首先将一幅图像分成帧图像或场图像两种类型。对于隔行视频序列，实际上存在两场，习惯称为偶场和奇场，在 MPEG-2 文档中分别称为：顶部场和底部场，下面分别简称它们为 T 场和 B 场，并定义它们的奇偶性，T 场的奇偶性为 0，B 场的奇偶性为 1。如果将 T 场和 B 场交叉后形成一帧图像作为独立的图像进行编码，这称为帧图像编码，如果每一场作为一幅独立图像进行编码，称为场图像编码。不管是帧图像还是场图像均可以采用 I、P 和 B 运动预测方式。

除了对帧图像的 16×16 的标准宏块预测方法外，MPEG-2 还支持以下几种运动预测和补偿方法。

（1）对场图像的场预测方式：假如有一组场图像序列为 $T_1B_1T_2B_2T_3B_3$，设 T_1 和 B_1 已经编码，现在分别编码 T_2 和 B_2，对 T_2 按宏块（MB）进行运动预测，如果 T_2 按 P 场进行预测，它可以选择来自 T_1 或 B_1 的一个宏块作为预测块，按一种优化准则来选择。如果 T_2 按 B 场进行预测，它可以从 T_1 或 B_1 选择前向预测宏块，从 T_3 或 B_3 选择后向预测宏块。对 B_2 场编码时，参考帧的选择原则与 T_2 一致。

（2）对帧图像的场预测方式：对帧图像中的一个宏块，按每一行是属于 T 场还是 B 场，分成两个 16×8 的块，如图 4.5 所示。每个 16×8 块单独进行运动估计和预测。每个块的运动预测方式类似于对场图像的场预测方式。

（3）场图像的 16× 8 运动预测方式：对场图像的宏块，分成上 16× 8 块和下 16×8 块，分别进行运动预测，对于 P 场预测，每个宏块传两个运动矢量，对于 B 场预测，每个宏块传 4 个运动矢量。

（4）双基预测方式：这种预测方式，仅对前向预测图像进行。对于一个宏块，将它分裂成两个 16×8 的场块，对每一个场块，用两个参考帧进行两次预测，将预测结果平均，作为最终的运动预测。对一个场块，首先进行的预测是在同奇偶性的场之间进行，如要预测的场块属于 T_2，以 T_1 为参考帧进行预测，并将该运动矢量作为该块的基本运动矢量，然后再以 B_1 为参考帧，进行运动估计和预测，对这个运动矢量只传送一个校正项，两次预测值的平均作为这个场块的最终运动预测值。双基预测在压缩效果上可以达到与 B 预测相当，又避免了 B 预测的附加延迟。

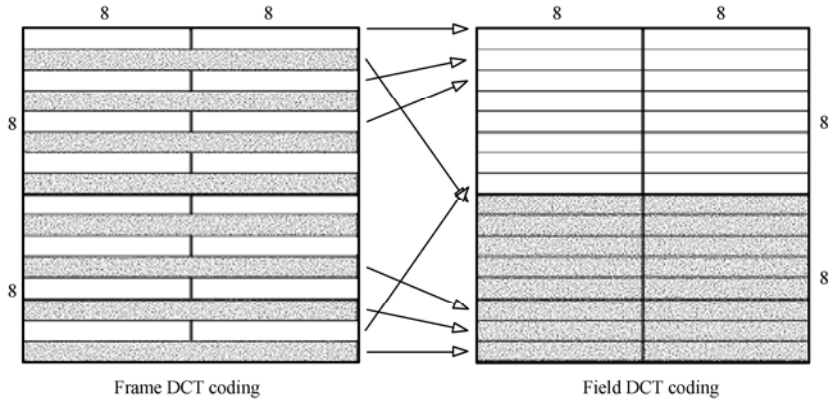


图 4.5　帧图像宏块的按场分裂

4.2.2　MPEG-2 的变换和扫描

对隔行视频，在帧图像中，相邻的行属于不同场，在景物中存在快速垂直运动时，相邻行的相关性降低，影响编码效率。MPEG-2 通过按帧和场进行 DCT 的方法解决这个问题。如图 4.5 所示的宏块，可以直接将这个宏块分成 4 块进行 DCT，这就是按帧 DCT，也可以如图 4.5（b）所示，将不同场的行抽出组成上部块和下部块，对上部和下部再分成两个块进行 DCT，这样，每个进行 DCT 的块中就是由同一场的相邻行组成的。

MPEG-2 也提供了第二种对量化后 DCT 系数的扫描方式：交替扫描（Alternate Scan），对于一幅完整的图像，可以选用其中一种扫描方式。两种扫描方式如图 4.6 所示，交替扫描比 Zig-Zag 扫描更先处理垂直高频分量。

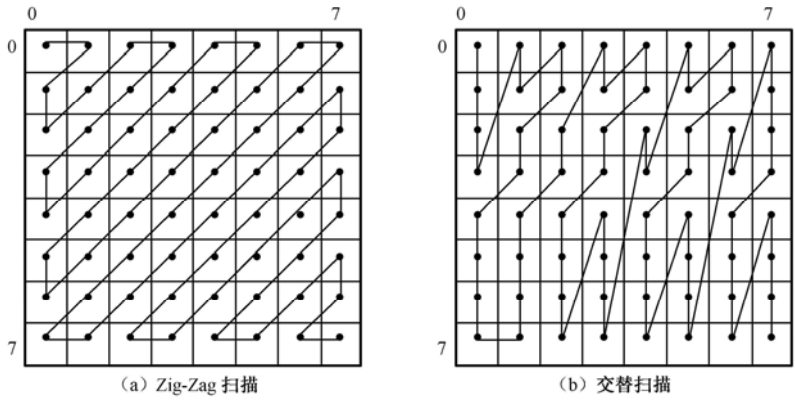


图 4.6　MPEG-2 的两种扫描方式

4.2.3　MPEG-2 的可分级编码模式

可分级编码模式是指在形成的一个码流中，只传输或解码其中部分码流，仍可以得到完整的解码图像，只是相对于全部解码整个码流，或降低了分辨率，或降低了质量，或减少了帧率。MPEG-1 的码流不具有这样的特性，如果只解码一帧图像一半的码流，可能只解码出上半部分图像，其他被丢失了。如果解码可分级的 MPEG-2 的一帧图像的一半码流，可以解码出的是完整的一帧图像，只是比解码全部码流的分辨率有降低。

一般可分级编码至少支持将码流分为基层和增强层，基层包含解码基本图像所需的码流，增强层可以补充为达到更高质量或分辨率或帧率所需的附加码流。可分级编码模式，可能会产生的问题是在编码器和解码器产生“偏移”，当在编码器使用包含增强层在内的高质量解码图像做参考帧，而在解码器只解码基层码流时，编码器和解码器的解码图像是不一致的，这就是偏移现象，当遇到一个新的 I 帧时，偏移置零。

MPEG-2 支持 4 种可分级编码模式。

（1）数据划分：将编码数据划分为至少两个层，例如，基层和增强层，将更重要的数据放入基层，如头信息、运动矢量，低频 DCT 系数等；将不太重要的系数放入增强层，如高频 DCT 系数。在解码端，如果只收到基层，则置高频系数为零，得到一个降低了质量的重构图像。在通信网中传输时，对基层进行重点保护，对增强层可以进行更低的保护。图 4.7 是数据划分的例子，数据划分会产生偏移现象。

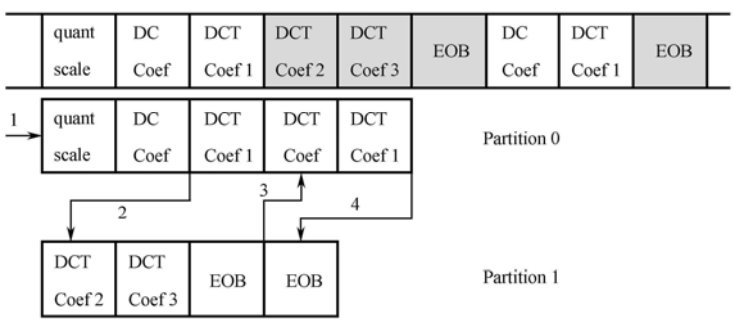


图 4.7　数据划分的示意图

（2）SNR 分级：SNR 分级是在 DCT 变换域进行，首先，DCT 变换系数通过量化器 Q1，它用粗的量化步长，产生的系数通过扫描和 VLC 得到基层码流。由原始 DCT 系数减去 Q1 的输出再通过 Q1 反量化器输出的值，该差值再通过量化器 Q2，Q2 的输出通过扫描和 VLC 产生增强层码流。在编码器，基层和增强层相加产生的重构图像作为后续图像的参考帧。在 4～9 b/s 范围内，用 SNR 可分级码流比不可分级码流损失 0.5～1.1 dB。SNR 分级也会产生偏移现象。

（3）空间分级：也就是空间分辨率分级，同样分成基层和增强层，基层是用对输入图像做亚采样后的低分辨率图像进行类似 MPEG-1 的编码。增强层用原始图像分辨率，预测图像是时域预测与空间预测的加权平均，其中时域预测是来自前一帧的已编码的全分辨率图像，与一般运动预测没有区别，空间预测是由基层解码的当前图像进行空间插值产生的，在每个宏块，加权系数可调整。在 4 Mb/s 码率下，空间分级比不可分级编码损失 0.75～1.5 dB。空间分级不产生偏移现象。

（4）时域分级：时域分级是首先均匀地丢掉一些帧或场，对保留的图像按标准方式编码，形成基层码流。丢掉的图像组成增强层，再对丢掉的图像用基层和增强层以编码帧作为参考图像进行运动预测补偿和变换编码。对比不可分级码流，时域分级只有 0.2～0.3 dB 的损失。时域分级也不产生偏移现象。

在 MPEG-2 中，方便应用的分级编码（除时域分级外）总会产生不可忽略的性能损失，这在选择应用时要根据系统需要谨慎处理。

4.2.4　MPEG-2 分档和分层

为了支持各种应用，MPEG-2 定义了各种档和层，档定义了各种实现工具，层定义了参数范围，档和层组合，支持不同的应用范围，表 4.5 列出了 MPEG-2 目前定义的各种档和层的组合。例如，档 Simple 支持 I，P 预测方式和 4∶2∶0 输入格式。前 5 档是分层继承的，例如，SNR 档除继承 Main 档的功能，还增加了 SNR 分级模式。

表 4.5　MPEG-2 的档和层组合

档（Profile）									
层			Simple （IP） 4∶2∶0	Main (IPB) 4∶2∶0	SNR (IPB) 4∶2∶0	Spatial (IPB) 4∶2∶0	High (IPB) 4∶2∶0	多目 (IPB) 4∶2∶0	4∶2∶2 (IPB) 4∶2∶0 4∶2∶2
	Low	点/行 行/帧 f/s Mb/s			352 288 30 4	352 288 30 4		352 288 30 8	
	Main	点/行 行/帧 f/s Mb/s	720 576 30 15	720 576 30 15	720 576 30 15		720 576 30 20	720 576 30 25	720 512 30 50
	High-1440	点/行 行/帧 f/s Mb/s	1 440 1 152 60 60		1 440 1 152 60 60	1 440 1 152 60 80	1 440 1 152 60 100		
	High	点/行 行/帧 f/s Mb/s	1 192 1 152 60 80			1 192 1 152 60 100	1 192 1 152 60 130	1 192 1 152 60 300	

档/层的组合针对不同的应用，例如，Main/Main 是 SDTV 和 DVD 所采用的模式，档和层的标志作为头数据，由解码器解码后判断是否有解码该码流的能力。

4.3　MPEG-4

为了适应多媒体通信的快速发展，ISO 于 1994 年开始制定 MPEG-4 标准。MPEG-4 最初是为了满足视频会议等的需要制定的，可以对音频、视频对象进行高效压缩，仅限于极低比特率的应用。后来经过不断发展成为一个可以适应各种多媒体应用，提供各种编码比特率的标准。

MPEG-4 视频编码的目标在于提供一种通用的编码标准，以适应不同的传输带宽、不同的图像尺寸和分辨率、不同的图像质量，进而为用户提供不同的服务，满足不同处理能力的显示终端和用户个性化的需求。

与传统的基于像素的视频压缩标准（如 MPEG-1、MPEG-2、H.261、H.263 等）不同，MPEG-4 采用基于对象的视频编码方法，不仅可以实现对视频图像数据的高效压缩，还可以提供基于内容的交互功能。除此之外，为了使压缩后的码流具有对于信道传输的鲁棒性，MPEG-4 提供了用于误码检测和误码恢复的一系列工具，这样采用 MPEG-4 标准压缩后的视频数据可以应用于带宽受限、易发生误码的网络环境中，如无线网络、Internet 和 PSTN 网等。

为了支持对多媒体内容的访问和操作，在 MPEG-4 标准中引入了 A/V（Audio/Visual Object）的概念，即音/视频对象。所谓对象就是在一个场景中能够被访问和操作的实体。在引入了 A/V 对象概念的基础上，MPEG-4 能够对 A/V 对象进行各种操作，增强了用户和对象之间的交互。MPEG-4 采用了基于对象的压缩编码方法，把视频分割成各种不同的对象实体，分别进行处理，针对不同的对象，进行比特流控制，并能实现多种基于对象的交互功能，有广泛的应用前景。

整个 MPEG-4 多媒体压缩标准协议主要分为 MPEG-4 系统，MPEG-4 音频，MPEG-4 视频等基本部分和扩充的部分。

4.3.1　MPEG-4 的组成

1. MPEG-4 系统流

MPEG-4 通信系统的整个工作过程是：首先由发送端压缩视听场景信息，并增加一些同步信息，然后将这些信息传递给一个传输层，再由传输层通过多路复合（Multiplex）技术将其打包成一个或多个用于传输或存储用的二进制码流：在接收端再将这些码流分解（Demultiplex）和解压缩，其中的视听对象将根据场景描述和同步信息被复合起来呈现给最终用户，最终用户可以有选择地与呈现的结果进行交互，这些交互信息可以在本地处理或发送回给发送端。

MPEG-4 由以下 6 个主要部分组成：

1）DMIF（The Delivery Multimedia Integration Framework）

DMIF 即多媒体传送整体框架，它主要解决交互网络中广播环境下及磁盘应用中多媒体应用的操作问题。通过传输多路合成比特信息来建立客户端和服务器端的交互和传输。通过 DMIF，MPEG-4 可以建立起具有特殊质量服务的信道和面向每个基本流的带宽。

2）数据平面

MPEG-4 中的数据平面可以分为两部分：传输相关部分和媒体相关部分。为了使基本流和 AV 对象在同一场景中出现，MPEG-4 采用了对象描述（OD）和流图桌面（SMT）的概念。OD 传输与特殊 AV 对象相关的基本流的信息流图。桌面把每一个流与一个 CAT（Channel Association Tag）相连，CAT 可以实现该流的顺序传输。

3）缓冲区管理和实时识别

MPEG-4 定义了一个系统解码模式（SDM），该解码模式描述了一种理想的处理比特流句法语义的解码装置，它要求特殊的缓冲区和实时模式，通过有效的管理，可以更好地利用有限的缓冲区空间。

4）音频编码

优越性在于不仅支持自然声音，而且支持合成声音。

5）视频编码

基于对象的视频编码，支持对自然和合成视觉对象的编码。

6）场景描述

MPEG-4 提供一系列工具，用于组成场景中的一组对象，一些必要的合成信息就组成了场景描述。这些场景描述以二进制格式表示，与 AV 对象一同传输、编码。场景描述主要用于描述各 AV 在同一具体 AV 场景坐标下如何组织与同步的问题，同时还有 AV 与 AV 场景的知识产权保护的问题。

MPEG-4 码流中的基本码流包括音频、视频和场景描述的编码表示。系统流和基本流可表示用于识别码流、描述逻辑相关性和内容描述的信息，每种码流都只包含一种数据类型。基本码流将用各自的解码器进行解码，视听对象根据场景描述信息复合并由终端显示设备显示出来，这些过程都是由系统解码模型中同步层所提供的同步信息同步的。

图 4.8 是 MPEG-4 系统的终端模型，它是对 MPEG-4 终端行为的一个抽象描述，是一种解码器模型。

2. MPEG-4 视频流

MPEG-4 标准的编码是基于对象的，这样就便于操作和控制对象，而传统压缩方法是基于帧的，无法对对象进行操作。MPEG-4 对比特率的控制可以基于对象，即使在低带宽时，也可以用码率分配方法，对用户感兴趣的对象可以多分配一些比特，而对于用户不感兴趣的对象可以少分配一些比特，这样图像的主观质量就得到了保证。

MPEG-4 中的对象操作使用户可以在用户端直接将不同的对象进行拼接，得到用户自己合成的图像，这在传统方法中也是无法实现的。

MEPG-4 在可分级性上也有很好的灵活性，可进行时域和空域的分级。在 MPEG-4 中，可根据现场带宽和误码率的客观条件，在时域或空域进行分级，时域分级是在带宽允许的情况下在基本层之上的增强层增加帧率，在带宽窄时减少帧率，达到充分利用带宽，使图像质量更好；空域分级是指对基本层中的图像进行采样或插值，增加或减少空间分辨率。

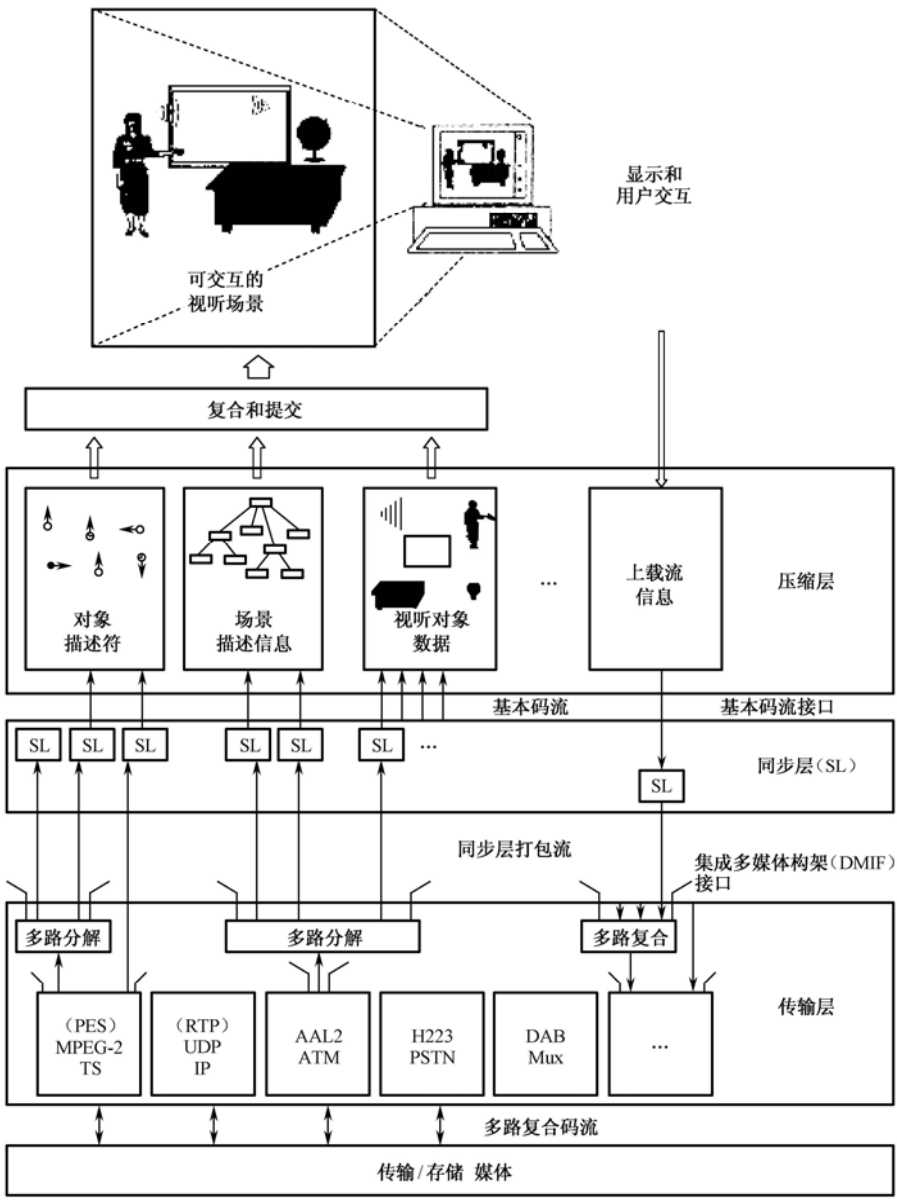


图 4.8　MPEG-4 系统的终端模型

为了支持前面提到的各种功能：高效压缩、基于内容的交互及基于内容的分级扩展，必

然要求 MPEG-4 要以基于内容的方式表示视频数据，因此，MEPG-4 引入了 VO（Video Object）的概念来实现基于内容的表示。VO 可以是一个矩形帧（即传统 MPEG-1，H.263 中的矩形帧），从而与原来的标准兼容；对于基于内容的表示要求较高的应用来说，VO 可以是场景中的某一物体或是某一层面。在 MPEG-4 中，VO 主要被定义为画面中分割出来的不同物体，每个 VO 有三类信息来描述：运动信息，形状信息和纹理信息。

VO 的生存期是一个镜头（Session），MPEG-4 首先对视频序列进行镜头切分，对一个镜头中的每一帧进行物体分割，得到各个 VO。

图 4.9 为一个 MPEG-4 的编码器框图。第一步是 VO 的生成，首先从原始视频序列中分割出 VO，之后由编码控制机制为不同的 VO 及各个 VO 的三类信息分配码率，之后各个 VO 分别独立编码，之后将各个 VO 的码流复合成一个位流。其中，在编码控制和复合阶段可以加入用户的交互控制或由智能化的算法进行控制。VO 的分割算法在 MPEG-4 中没有定义，随着研究的进展，可以选用更好的分割算法。

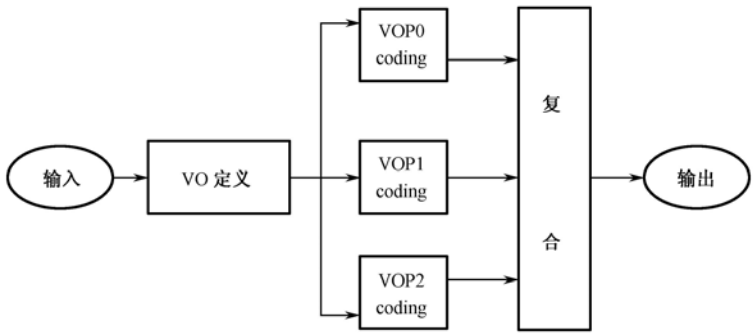


图 4.9　MPEG-4 编码器原理框图

4.3.2　MPEG-4 视频编码原理

在数字域表示图像/视频的方法通常是基于像素的表征方法，目前使用的主要技术都依赖于这种表征方式。在 MPEG-4 标准之前的 MPEG-1、MPEG-2、H.261 和 H.263 标准等都是采用传统的图像编码方法，依据 Shannon 信源编码理论的框架，将图像作为随机信号，利用其统计特性来达到压缩的目的。这种方法把视频序列按时间先后分为一系列的帧，每一帧的图像又分成宏块以进行运动补偿和编码，这种基于帧、块和像素的编码被称为第一代视频编码方案。

第一代视频编码的核心技术是基于分块的离散余弦变换（DCT），但是在低比特率应用环境下，压缩图像不可避免地会出现块效应。这是因为采用平稳高斯过程来表示非平稳的图像信号，用余弦基作为非平稳信号的逼近，其结果必然不是最优的。另外，由于这种方案本身未能考虑信息获取者的主观特性及图像的具体结构和内容，也没有充分利用人类视觉系统的特性，难以实现对图像内容的查询、处理等操作。

20 世纪 80 年代中期，受到人眼视觉系统生理机制的启发，出现了一些新的视频图像表征方法。这种方法利用现代图像编码方法和人眼视觉特性，从轮廓、纹理的思路出发，支持基于内容的交互功能。目前这种基于对象的第二代视频编码的概念已被人们广泛地接受，并

且已经成为新的图像/视频编码标准——MPEG-4 的基础。

在基于对象的视频编码中，编码的基本单元是视频对象。为了实现基于对象的表示，MPEG-4 定义了视频对象（Video Object，VO）和视频对象平面（Video Object Plane，VOP）等概念。根据人眼感兴趣的一些特性，如纹理、运动、形状等，将视频序列的每一帧通过人工、半自动或全自动等方式分割为一些任意形状的图像区域，这些区域被称为视频对象平面，同一对象连续的 VOP 称为视频对象。

图 4.10 所示的是 MPEG-4 视频编码原理框图，从图中可以看出，MPEG-4 的视频编码就是针对运动信息、形状信息和纹理信息 3 种信息的编码技术，其编码过程如下所述。

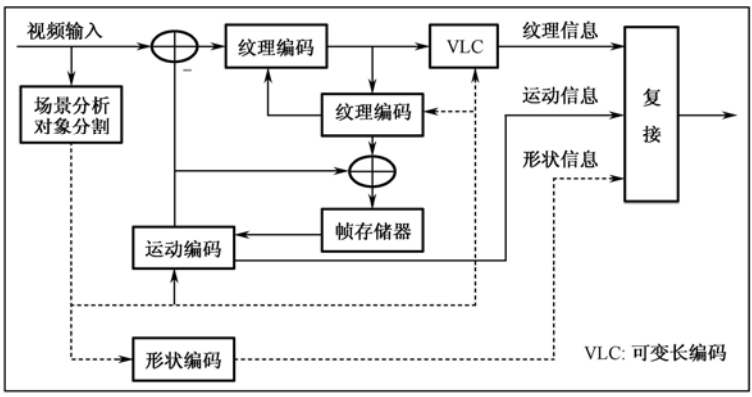


图 4.10　MPEG-4 视频编码原理框图

（1）从原始视频流中采用自动、半自动、人工等方法分割出 VO。

（2）对 VO 进行编码，对不同 VO 的运动信息、形状信息和纹理信息分配不同的码字。

（3）将各个 VO 的码流复合成符合 MPEG-4 标准的比特流。

其中，纹理编码、运动预测和运动补偿部分在原理上同现有标准基本一致，而形状编码是视频编码标准中第一次引入的技术。

由于 MPEG-4 标准过于复杂，下一节我们结合 MPEG-4 的校验模型，分析 MPEG-4 视频编码的一个有代表性的简单的实现方法，也可以通过对比，发现它们与 MPEG-1、MPEG-2 相比的实质性变化。

4.3.3　MPEG-4 中视频编码器的实现

MPEG-4 编码器可以对图像序列中具有任意形状的 VOP 进行编码，对 VOP 的编码就是针对某一时刻帧画面 VO 的形状、运动、纹理等信息进行编码。

1. 形状编码

相对于以前的压缩标准来说，MPEG-4 标准第一次引入了形状编码算法。

在 MPEG-4 标准中，VO 的形状信息有两类：二值形状信息和灰度形状信息，两种信息都可以采用位图法表示，位图法具有较高的编码效率和较低的运算复杂度。

二值形状信息用 0、1 来表示 VOP 的形状，0 表示该像素在 VOP 区域之外，1 表示在

VOP 区域之内。对二值形状信息进行编码时可以采用基于块的运动补偿技术，可以是无损编码，也可以是有损编码。灰度形状信息用 0～255 之间的数值表示该像素的透明度，编码时采用基于块的运动补偿 DCT 方法，属于有损编码。引入灰度级信息主要是为了使前景物体叠加到背景上时不至于边界太明显，太生硬。

形状编码的关键是 VOP 的生成，这涉及图像分割方面的问题，不在 MPEG-4 标准的研究范围之内，在基于对象的 MPEG-4 视频编码中，输入序列就已经包含了表示 VOP 形状信息的 MASK 序列信号。

在已知 VOP 的情况下，校验模型采用位图法表示形状信息，VOP 被一个边框框住，边框的长，宽均为 16 的整数倍，同时保证边框最小。这种位图法实际上是将二值形状编码转换为一个边框矩阵的编码，具体采用的是基于上下文的算术编码（Context-based Arithmetic Encoder，CAE）方法。

形状编码过程框图如图 4.11 所示。

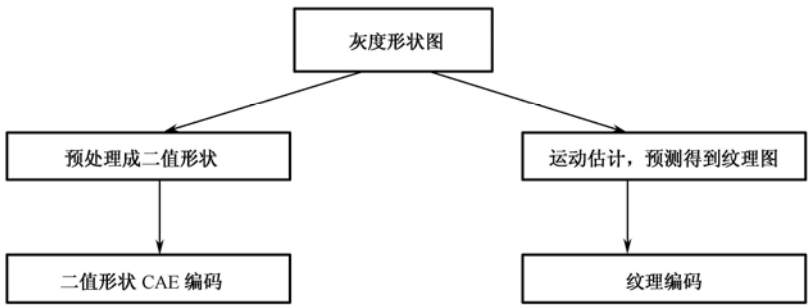


图 4.11　形状编码过程框图

2. 运动信息编码

MPEG-4 利用运动估计和补偿技术来去除帧间的时间冗余度。主要区别在于其他标准中采用的是基于块的技术，而 MPEG-4 中采用的是 VOP 结构。VOP 有 3 种编码模式：帧内编码模式（I-VOP）、帧间编码模式（P-VOP）和帧间双向预测编码模式（B-VOP）。很显然，只有对 P-VOP 和 B-VOP 编码时才需要运动估计。

如果一个宏块完全位于一个 VOP 内，就采用一般的基于 16×16 像素块或 8×8 像素块进行运动估计，运动矢量以半像素精度进行计算。如果一个宏块只有一部分位于 VOP 内，则采用修正的块匹配技术估计运动矢量。当参考块位于 VOP 边界上时，采用重复填补技术给位于 VOP 外的那些像素指定值，然后利用填补后的像素值估计运动矢量。这样在 VOP 的边界上搜索预测所需的候选像素时就有了更多的选择，从而提高了效率。

与 H.263 类似，MPEG-4 也支持重叠运动补偿，在更低比特率的情况下可以获得更好的预测质量。

- **重复填充技术**

在 MPEG-4 中存在对任意形状 VOP 进行编码的问题，所以在 VOP 是边缘块运动估计的时候，就不是矩形匹配，而是多边形匹配了，于是，参考 VOP 需要采用基于宏块的重复填充技术，即使是基于矩形帧的编码，由于分块编码的原因，也需要将该帧的边长扩充至 16

的整数倍，因此，基于宏块的重复填充技术是必要的。

在基于 MPEG-4 简单框架编码中，边界扩展技术如下：边界处的点可以直接用相邻的水平或垂直边缘块来填充，但是如果一个外部宏块和多于一个内部宏块相邻，那么采用最大优先数的宏块进行填充。一个外部宏块的四周宏块的优先顺序为下、右、上、左。剩余的不和任何内部宏块相邻的宏块用 128 来填充。

- **多边形匹配**

多边形匹配主要利用形状信息，对一个宏块内而且是 VO 内部的点才计算在内，对外部点就不计算，对于 VOP 边缘宏块就采用多边形匹配。

多边形匹配的公式：

$$\mathrm{SAD}_N(x,y)=\sum_{i=1,j=1}^{N,N}|\mathrm{original}-\mathrm{previous}|(\neg(\mathrm{Alpha}_{\mathrm{original}}=0))$$

- **半像素搜索**

首先对参考的 VOP 进行双线性插值，然后进行运动矢量的搜索，这样能够得到比较准确的运动估计。图 4.12 是双线性插值的示意。

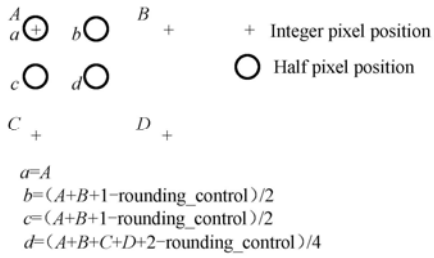


图 4.12　双线性插值示意图

- **重叠的运动预测**

每个 8×8 的亮度块中的像素值是通过三个预测值加权和得到的，使用三个运动矢量就可以得到三个预测值：有当前亮度块的运动矢量，下面四个“远程”运动矢量中的两个：

当前亮度块左边和右边块的运动矢量；

当前亮度块上面和下面块的运动矢量。

- **Inter/Intra 编码方式的选择**

在像素的运动估计后，编码器决定宏块采用 Inter 还是 Intra 的编码方式，具体的判断准则如下：

$$\mathrm{MB\_mean}=(\sum_{i=1,j=1}^{Nc}\mathrm{original})/N_c$$

$$A=\sum_{i=1,j=1}^{16}|\mathrm{orignal}-\mathrm{MB\_mean}|(\neg(\mathrm{Alpha}_{\mathrm{orignal}}=0))$$

$$N_B=Nc\times 2^{(\mathrm{bit\_per\_pixel-8})}$$

其中，Nc 是 VOP 内部的点数。

如果 $A<(\mathrm{SAD}_{\mathrm{inter}}-2N_B)$，就采用 Intra 方式编码，不必进行运动搜索；否则采用 Inter 方式，然后继续进行半像素的运动搜索。

3. 纹理编码

VOP 的纹理信息包含在视频信号的亮度分量和两个色度分量中。对于 I-VOP，纹理信息直接包含在亮度和色度分量中，而对于运动补偿后的 VOP，纹理信息包含在运动补偿后的残差中。对纹理信息编码时，采用标准的基于 88 像素块的 DCT 方法。完全属于 VOP 内部的 88 块直接进行编码，而对于横跨 VOP 边界的块，先采用图像填充技术得到 VOP 之外的像素值，再进行编码。纹理编码过程如图 4.13 所示。

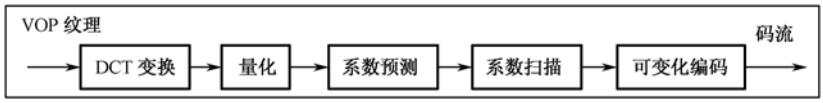


图 4.13　纹理编码过程

4. 分级扩展编码

在远程多媒体数据库检索、视频内容重放及视频通信等应用中，可分级扩展编码的引入使得译码端可以依据具体的信道带宽、系统处理能力、显示能力及用户需求进行多分辨率的解码及重放，从而提供了灵活的编码策略。

MPEG-4 提供了两种可分级扩展方式：时间可分级扩展和空间可分级扩展。时间可分级扩展编码是用增强层来增加 VOP 基本层的时间分辨率，编码时每隔几帧选一帧作为基本帧，基本帧必须编码传输。带宽允许或译码端要求时才编码中间帧，这些中间帧作为增强层用来增加基本帧的时间分辨率，使运动图像看起来更加平滑。而空间可分级扩展是用增强层来增加 VOP 基本层的空间分辨率，在编码端基本层对原图像进行亚采样，使图像的尺寸变小，在译码端再通过插值恢复原来图像的大小，增强层则提供对原分辨率的质量改善。

5. Sprite 编码

Sprite 对象是针对背景对象的特点提出来的。通常情况下背景对象本身没有运动，通过图像的镶嵌技术把整个序列的背景图像拼接成一个大的完整的背景图像，这个图像叫做 Sprite 图像。Sprite 图像只需要编码传输一次并存储在解码端，随后的图像可以从 Sprite 上恢复所有图像的背景。MPEG-4 中包括 Sprite 是因为这种编码方式可以提供很高的压缩效率。

基于 Sprite 的编码非常适合于合成对象，也可以用在发生了剧烈运动的自然场景中。为了支持低处理延时的应用，传输 Sprite 时可以采用多种方法。图 4.14 是一种实现 Sprite 编码的示意图。

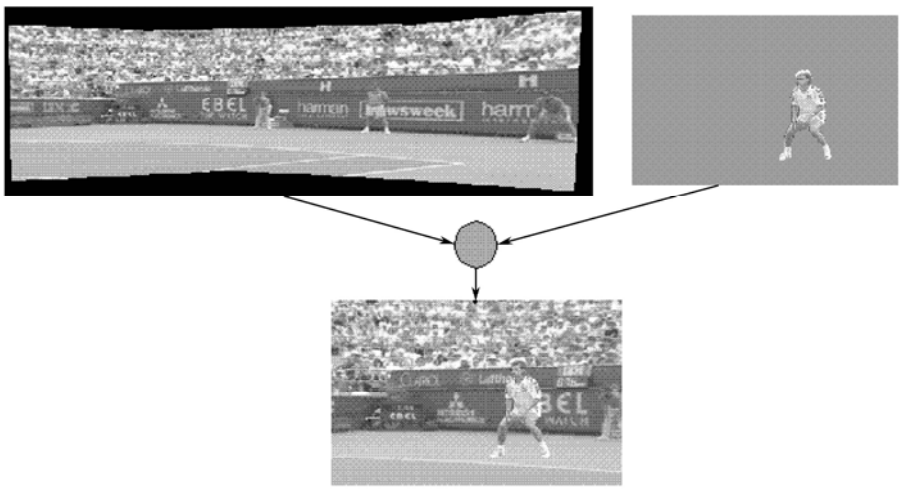


图 4.14　Sprite 编码的示意图

4.3.4　MPEG-4 中的差错控制方法

随着移动通信的迅猛发展，无线网络上的视频传输图像受到广泛重视。由于多径衰落、时延扩展、噪声影响和多址干扰等原因，无线信道误码率比较高，一般在 10^{-5} 以上，有的可以高达 10^{-2}。MPEG-4 通过去除时间冗余度来达到压缩的目的，而高度压缩后的视频码流对传输时所产生的误码非常敏感，一旦发生了误码，不仅影响该误码数据的恢复，还会影响与之相关的其他数据的恢复，造成“误码扩散”，使恢复出来的信号面目全非。为了使压缩后的码流具有对于信道传输的鲁棒性，MPEG-4 提供了在各种无线和有线低码率网络下可靠传输图像的方法。

1. MPEG-4 中的差错控制方法

MPEG-4 依据自身定义的“视频包”这一码流结构来进行误码检测，如图 4.15 所示，一个视频包由一个重同步码、一个报文头和宏块数据组成。重同步码是一个唯一的码字，由一系列的 0 跟随一个 1 组成，报文头包含的信息可以帮助译码器重新开始译码处理，宏块数据由运动矢量、DCT 系数和视频包中包含宏块的编码模式信息组成。

图 4.15　视频包的数据结构

MPEG-4 中的差错控制方法有重同步标记、数据分割、头扩展字和可逆变长编码等。

1）重同步标记（Resynchronization Marker）

MPEG-4 编码时采用了可变长编码（VLC），码字的分界并不明确，传输的错误会导致编

码器同步信息的丢失。一旦检测到误码，一般在两个重同步指针之间的所有数据都会被丢弃。MPEG-4 在码流中插入了重同步码标记，在发生了误码的情况下，译码器会跳到下一个重同步标记处，并在此标记处重新开始译码。

2）数据分割（Data Partitioning）

数据分割方法将视频包内的运动信息、纹理信息等宏块数据分开。如果只有纹理信息丢失，数据分割允许使用运动信息以更为有效的方式进行误码掩盖。这样，数据分割就提供了一种从损坏的视频包中恢复更多数据的机制。

3）头扩展码（Head Extension Code）

包含在视频帧中的一些固定的重要信息，比如视频数据的空间尺寸，与译码有关的时间标记、对视频数据的描述和当前帧的类型（帧内或帧间编码）等，在视频数据开始的信息头中被传送。为了减少这一数据信息的敏感度，MPEG-4 的报头信息中引入了 1 比特的头扩展码字（HEC）域。使用 HEC 极大地减少了头信息被破坏的概率，从而获得一个更高质量的译码视频。

4）可逆变长编码（Reversible Variable Length Coding）

可逆变长编码可以进一步降低译码数据中的错误对恢复视频的影响。可逆变长编码是可以采用前向和后向两种方式进行译码的码字，一旦出现误码，即使跳到了码流中的下一个重同步标记处，可逆变长编码仍然可以从下一个标志处对受损的部分码流进行反向译码，从而限制误码的影响。

2. 不等的错误保护

MPEG-4 提供的抗误码工具，在误码率低于 10^{-5} 时可以使图像的重建质量达到可接受的水平，然而当误码率比较高时，就需要采用信道编码来改善信道状况。通过采用信源和信道联合编码技术，在误码率较高的无线信道上传输并获得可接受的视频重建质量是可能的。

我们知道，MPEG-4 码流的不同部分对于图像重建质量的作用是不同的。这样的码流结构，非常适合于采用不等的错误保护机制，从而实现信源与信道的联合编码。因此，针对 MPEG-4 视频压缩码流的结构，在分配给信源编码的比特率一定的情况下，可以结合信道编码技术，根据视频码流各部分的重要性程度采用不等的错误保护，比如对于重要的信息采用纠错能力较强的信道编码方法，而对于其他不重要的信息采用纠错能力较弱的编码方法，使得视频的重建质量最高。图 4.16 中所示的是 MPEG-4 视频包采用不等的错误保护机制的一个例子，其中，$r_1 < r_2 < r_3$。

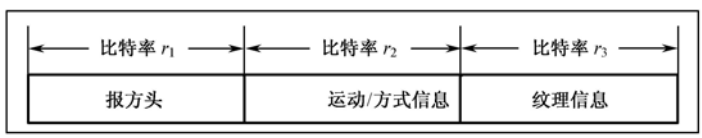


图 4.16　不等的错误保护机制

报文头是视频包中最重要的部分，对它采取的保护也最多，运动部分将获得低一级的保护。由于即使没有纹理部分，译码器依然可以通过运动补偿对误码进行掩盖而不会使重建图像质量有太多的下降，所以纹理部分获得最低水平的保护。采用这样的方法，视频包的重要部分就不太可能发生误码。

4.3.5　MPEG-4 中的解码技术

1. 视频对象平面解码器结构

解码器主要是由两部分组成，如图 4.17 所示。形状解码器和传统的运动和纹理解码器。重构的 VOP 是由形状、纹理和运动信息正确组合而成的。

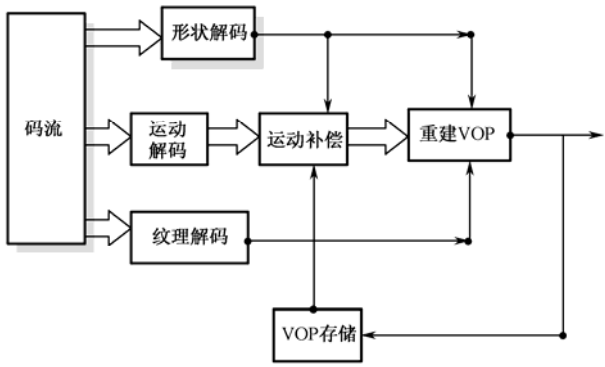


图 4.17　解码结构框图

2. 时域预测结构

（1）解码 P-VOP 时，应参考最近已经解码的 I-VOP 或者 P-VOP。I-VOP 和 P-VOP 是否编码由标志 VOP-Coded 指示，如果为 0 表示没有编码。如果最近邻的 I-VOP 或者 P-VOP 的 VOP-Coded 标志为 0，那么当前的 P-VOP 应该参考次近邻的 I-VOP 或者 P-VOP，直到 VOP-Coded 不为 0。

（2）解码 B-VOP 时，应参考解码的前向和后向的 VOP，这些要参考的 VOP 的 VOP-Coded 标志不为 0，如果前向和后向 VOP 的 VOP-Coded 为 0，那么使用以下规则：

对于纹理，目标 B-VOP 的预测值是一个宏块，它的值为（Y,U,V）=(128,128,128)。

对于二值 alpha 平面，预测值为 0。

对于灰度级的 alpha 平面，预测值为 128。

时域预测结构如图 4.18 所示。

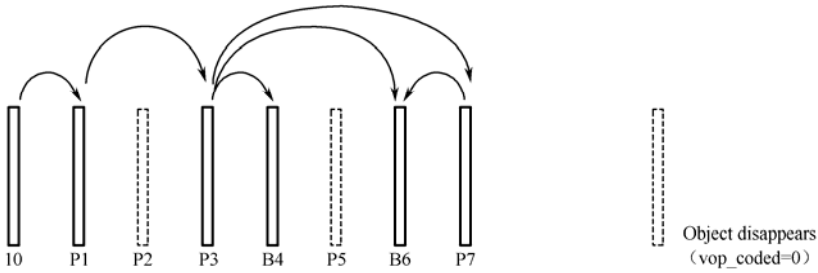


图 4.18　时域预测结构图

3. 合成器定义

解码器输出的重构 VOP 将成为合成器的输入，在合成器中 VOP 按 VOP-composition_order 标志指定的顺序混合。每个 VOP 都有自己的 YUV 和 alpha 值，混合是按顺序进行的。例如，如果 VOP N 覆盖到 VOP M 上而产生新的 VOP P，那么合成的 Y,U,V 和 alpha 值为：

$P_{yuv}=((255-N_{alpha})\ M_{yuv}+(N_{alpha}N_{yuv}))/255$

$P_{alpha}=255$

如果一个特殊的序列不止两个 VOP，这个混合过程将对 YUV 分量反复使用，并将输出图像作为背景。

4.4　基于 MS320C62x 的 MPEG-2 视频解码器实现

本节介绍基于 TMS320C62x 系列 DSP 平台的 MPEG-2 视频解码器实现。MPEG-2 视频标准广泛应用在 DVB、DTV、DVD 和 DSS 等数字视频系统中。此解码器软件适用于 MPEG-2 配置为主类主级的电视信号格式，同时为了提高复用性，软件还遵照了 eXpressDSP 软件算法标准 xDAIS。介绍包括该解码器软件的不同方面，有算法概要、译码指南、解码器应用编程接口（APIs）、内存需求和性能等。

4.4.1　软件实现概述

MPEG-2 视频标准为具有娱乐品质的数字视频信号规定的解压缩和编码表示方法。它被广泛运用到各种数字视频信号系统中，如 DTV（数字电视）、DVB（数字视频广播）、DSS（卫星直播系统）和 DVD（数字多功能光碟）等。MPEG-2 视频解码器在电子消费领域（如 DVD 播放器、机顶盒及卫星直播系统单元等）扮演着一个重要的角色。面对不同的应用对象，软件实现能够比硬件实现更灵活、简单地对产品进行定制，而且也更容易根据新的特征进行升级。同时，可编程器件具有将多种复杂功能在同一个硬件平台上实现的优势，比如视频解码、调制解调功能和语言控制接口等。

目前，已经实现了 MPEG-2 主类主级视频解码器，它的最大输入速率为 15 Mb/s，色度格式采用最常见的 4∶2∶0，这种格式目前正在被很多应用领域采用。

4.4.2　算法描述

图 4.19 给出了 MPEG-2 视频解码算法的流程图，在保证较好的视频质量的前提下，MPEG-2 标准采用了一些技术来保持高的压缩比。

1. 采用运动补偿的帧间编码

运动补偿利用这样一个事实达到压缩：在一小段图像序列里面，场景都是相似的而且很多物体只移动了很小一段距离，利用这些时间冗余，当前画面的很多部分能够根据先前的解码结果进行预测。在运动补偿中，整个画面被划分成很多块，我们利用计算出来的二维空间运动矢量，以及先前解码出来的图像中的相应块，来预测当前画面中相应块对应的像素值，最后压缩不是通过像素分块来实现，而是通过对运动矢量和预测误差进行编码来实现的。预测误差拥有很小的空间冗余，而且能够用变换编码有效地进行压缩。

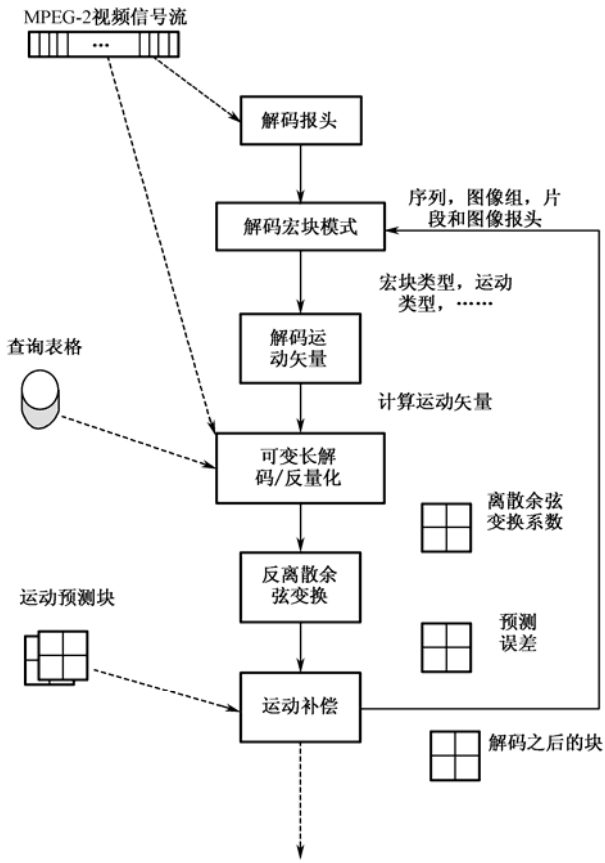


图 4.19　MPEG-2 视频解码算法流程图

2. 采用离散余弦变换的变换编码

在编码过程中，离散余弦变换（DCT）的对象是帧间编码宏模块的预测误差和帧内编码宏模块的像素值。图像被分割成 8×8 像素的小块，离散余弦变换将原始图像中的小块转换

成同等大小的、由水平和垂直方向的空间频率分量表征的块。虽然原始图像和预测误差的能量在一个分块里是随机分配的，但是经过 DCT 变换后，能量就主要集中到低频段了。我们用一个步长可变的量化器对频率分量进行量化来达到压缩图像的目的，该量化器的量化步长是根据视觉心理学来确定的，这样产生的量化误差就不容易被察觉。同时，很多高频分量系数非常小，它们经过量化之后的结果为零，利用这一特性，可以将量化后分块中的系数以“Z”字形顺序排成一串，这样可以得到最多的连续零串，然后采用行程编码，将一个分块编码成连续零个数和级别对的序列。

3. 可变长度编码

可变长度编码（VLC）根据每个行程级别对的发生频率来给它们分配码字。发生频率较高的行程级别对被分配的码字较短，相反发生频率较低的行程级别对被分配的码字较长。可变长度编码就是因为对占支配地位的高频率行程级别对赋予短码字来实现信号压缩的。

4.4.3　解码器的实现

在这一部分，我们将介绍实现的视频解码器的各个方面。

1）特点

以下是视频解码器软件的特点：

（1）整个视频解码过程是基于软件的，运行在可编程的 DSP 上，不需要任何硬件支持，这最大地保证了灵活性。

（2）此款解码器完全适应 MPEG-2 的主类主级规格，我们用官方提供的 MPEG-2 符合性测试流对解码器进行了严格测试，测试结果证实我们设计的解码器适应测试规范，这既保证了算法执行的正确性，又保证了输出图像的高质量。

（3）此款解码器还能处理 MPEG-1 参数受限的比特流（CPB）。

（4）此款解码器适应软件算法标准 xDAIS［3］。我们实现了所有的 xDAIS 规则和大部分指南要求。这保证了解码器的算法能够很容易地集成到不同的框架系统和环境。注意：xDAIS 算法标准本身有可能改动，软件也将升级以适应这些改动。

（5）此解码器是多信道使能的。该解码器是可再入的，而且能够同时处理几条不同的译码信道（此功能还需要进一步测试）。除非软件正在处理流水线代码，解码器能够在任何地方中断，当主频为 250 MHz 时，中断响应时间小于 10 μs。

2）解码器的结构

解码器分成了以下几个模块（参见图 4.20）。

- VLD，功能包括执行变长译码，行程展开和反量化。
- IDCT，功能是执行反离散余弦变换。
- 运动补偿地址计算，功能是计算参考块的位置并将它们存入存储器。
- 运动补偿的内核，计算预测像素值。
- 辅助功能，解码报头信息和运动矢量等。
- 执行 xDAIS 要求的 IALG 和 IRTC 之间的接口。

以上的模块是通过解码器控制代码组合到一起的。控制代码通过调用不同模块的函数来传递和接收数据。

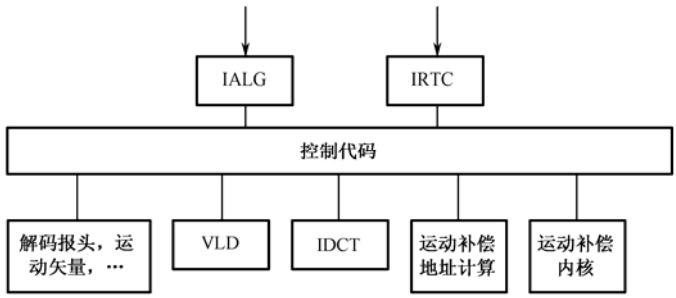


图 4.20　MPEG-2 视频解码器结构框图

3）译码指南

解码器程序是由 C 语言和 TMS320C62x 汇编语言组合执行的，代码遵循了所有的 xDAIS 规则，其中有些规则是：

- 解码器是可再入的。
- 所有相关的数据是完全可再定位的，同时，所有的解码器代码也要是完全可再定位的。
- 所有的外部定义都要以 MPEG-2VDEC 或 MPEG-2VDEC_ti 为前缀。

请参考 xDAIS 相关文档获得所有的编码规则。

4）中断问题

除非软件正在运行流水线环路，解码器能够在任何地方被中断。正如 xDAIS 规则中推荐的一样，当主频为 250 MHz 时，解码器的最长中断响应时间也小于 10 μs。

5）多信道时的执行

此解码器是可再入的，能够被应用在多信道环境下。每条信道的解码信息被保留在算法实例化对象 MPEG-2VDEC_Handle 中。用户或者框架利用解码器的应用编程接口 MPEG-2VDEC_create 为每一条解码信道创建一个算法实例化对象，创建完成以后，框架就能将实例化对象传送到应用编程接口 MPEG-2VDEC_ apply，进而对图像进行解码。

4.4.4　与解码器的连接

解码器能够设置成在自己内部运行或者连接到一些外界的框架系统中。当设置成后者时，框架能够利用解码器的应用编程接口（APIs）与解码器相连。在这一部分，我们介绍解码器的 APIs。另外，用一个框架代码示例来说明这种连接。

1. 解码器的应用编程接口（APIs）

解码器的 APIs 包括：

- MPEG2VDEC_init

```
Void MPEG2VDEC_init(Void);

Parameters    /*  参量  */
NULL.         /*  空  */

Return Value /*  返回值  */
NULL.        /*  空  */
```

功能描述：解码器的初始化，应该第一个被解码器调用

- MPEG2VDEC_create

```
MPEG2VDEC_Handle   MPEG2VDEC_create(
                        const   IMPEG2VDEC_Fxns *fxns,
                        const   MPEG2VDEC_Params *prms);
```

Parameters

参数	含义
const IMPEG2VDEC_Fxns *fxns	函数表
const MPEG2VDEC_Params *prms	创建参数

```
Return Value          /* 返回值 */
MPEG2VDEC_Handle     /* 算法例程控制 */
```

功能描述：创建一个算法实例化对象，每一个解码信道都要调用。

- MPEG2VDEC_delete

```
Void MPEG2VDEC_delete(MPEG2VDEC_Handle handle);

Parameters
```

参数	含义
MPEG2VDEC_Handle handle	MPEG2VDEC 算法例程手柄

```
Return Value   /* 返回值 */
NULL.          /* 空 */
```

功能描述： 删除一个算法实例化对象，完成一个解码信道工作后调用。

- MPEG2VDEC_exit

```
Void MPEG2VDEC_exit(Void);

Parameters  /*  参量  */
NULL.       /*  空  */

Return Value  /* 返回值 */
```

```
        NULL.        /*  空  */

        功能描述：  解码器退出。
```

- MPEG2VDEC_apply

```
        Void MPEG2VDEC_apply(MPEG2VDEC_Handle handle,
                             Int *input[],Int *output[]);

        Parameters
```

参数	含义
MPEG2VDEC_Handle handle	MPEG2VDEC 算法例程手柄
Int *input[1]	函数代码地址： functionCode. The functionCode 应该是 FUNC_DECODE_FRAME 或 FUNC_START_PARA
Int *input[2]	外部输入码流缓冲器起点地址
Int *input[3]	外部输入码流缓冲器大小
Int *output[1]	输出参数缓冲器地址
Int *output[2]	外部输出缓冲器地址
Others	保留用于特殊目的

```
        Return Value   /* 返回值 */
        NULL.         /* 空 */

        功能描述：用解码器来接收输入码流并将结果输出到输出缓冲器中。
```

框架应该调用应用编程接口函数 MPEG-2VDEC_init 来初始化解码器。初始化之后，架构应调用 MPEG-2VDEC_create 函数来为每一个信道创建一个算法实例化对象，这些算法实例化对象包含了对应解码信道的所有状态信息。然后，架构就可以调用 MPEG-2VDEC_apply 函数，利用解码器处理输入比特流。

1）输入原始数据

每当调用 MPEG-2VDEC_apply 函数时，框架就能将 MPEG-2 原始输入数据通过以参量 input[2]开头的输入缓冲器传送到解码器中，这个输入缓冲器的大小由参量 input[3]决定，并被设计成循环的形式。框架负责注满这个缓冲器和确保输入的数据足够解码一幅图像。通过一个指向该循环输入缓冲器头的指针变量((DECODE_OUT *) (output[1]) ->next_wptr)，框架能够知道有多少数据已经被解码器处理掉。这个输入缓冲器的大小必须是 4 B 的整数倍，并且要以 4 B 为边界进行排列。推荐将输入缓冲器的大小设置为 512 KB 或者更大，设置地点在外存储器中。

2）输出解码图像

MPEG-2VDEC 运算法则将解码后的图像输出到帧缓冲器中，该缓冲器的指针是 MPEG-2VDEC_apply 函数返回的参量 output[2]。算法要求保持 4 帧输出图像，所以输出帧缓冲器至少要有 4×图像尺寸的空间，此处图像尺寸=图像高度×图像宽度×1.5。此外，输出

帧缓冲器应该以 4 B 为边界进行排列。推荐输出缓冲器的大小为 2 440 KB，这样能很好地处理尺寸、格式分别为 720×576 和 4∶2∶0 的视频信号。

那些输出图像以 4∶2∶0 YUV 的格式存储。每当算法返回值时，客户机程序应当查看变量((DECODE_OUT *) (output[1]) ->outputting)，以确认解码图像是否准备好，如果准备好了，就能从存储单元((DECODE_OUT *) (output[1]) ->outframe)里找到输出图像。输出图像一直以帧为格式，这是为了避免在解码交织的视频信号时出现混乱，因为在那种情况下，同一个序列里的输出图像，格式可能是帧，也可能是扫描场。

3）输出参量

解码器返回的输出参量在 MPEG-2VDEC_apply 函数的返回参量 output[1]里。这个参量可能是视频序列开头的结构体 START_OUT，也可能是序列最后的结构体 DECODE_OUT，详见下面的代码实例。

```
/*********************************************************/
/***** Output parameter at the beginning of sequence. ****/
/*********************************************************/
typedef struct _START_OUT {
  Int fault;                          /* Any problem occur? */
  Int ld_mpeg-2;                       /* MPEG-2 stream? */
  Int bit_rate;                       /* Input bit rate */
  Int picture_rate;                    /* Output picture rate */
  Int vertical_size;                   /* Ori. pic. dimension */
  Int horizontal_size;
  Int coded_picture_width;             /* Coded pic. dimension */
  Int coded_picture_height;
  Int chroma_format;
  Int chrom_width;
  Int prog_seq;                       /* Progressive seq.? */
} START_OUT;
/*********************************************************/
/************Output parameter afterward.***************** /
/*********************************************************/
typedef struct _DECODE_OUT {
  Int fault;                          /* Any problem occur? */
  Int pict_type;                       /* I, P or B-pic. */
  Int pict_struct;
  Int next_wptr;                       /* Head of cir. input */
  Int topfirst;
  Int end_of_seq;                      /* End of sequence? */
  Int outputting;                      /* Output any frame? */
  SmUns outframe;                      /* Starting of out pic. */
} DECODE_OUT;
```

2. 框架代码实例

下面以一个示例框架代码为例来说明解码器的应用编程接口（APIs）的用法。

```
        #define SHARE_MPEG-2_RDBUF_SIZE      (128 * 1024)
        unsigned int share_bsbuf_storage[SHARE_MPEG-2_RDBUF_SIZE];
        /* 512KB input buffer. 4-bytes alignment. */
        #define MAX_PICT_SIZE              0x098800
        /* 610KB. Enough for 720×576 4∶2∶0. */
        #define NO_OF_FRAME_BUF            4
        unsigned char frame_all_storage[NO_OF_FRAME_BUF * MAX_PICT_SIZE];
        /* Output buffer. 4-bytes alignment. */
        #define MAXPARAM                  5
        int *in[MAXPARAM];
        int *out[MAXPARAM];
        int h_share_MPEG-2_rdbuf_size = SHARE_MPEG-2_RDBUF_SIZE;
        int functionCode;
                  …
        MPEG-2VDEC_Handle MPEG-2vdec;
        /*********************************************************/
        /* To do -- fill up the whole input buffer                */
        /*********************************************************/
        old_ptr = 0; /* head of circular buffer */
        MPEG-2VDEC_init();
        MPEG-2vdec = MPEG-2VDEC_create(&MPEG-2VDEC_TI_IMPEG-2VDEC, NULL);
        in[1]=&functionCode;                    out[1]=(int*)&out_para[0];
        in[2]  =  (int  *)  &share_bsbuf_storage[0];  out[2]  =  (int  *)
&frame_all_storage[0];
        in[3] = &h_share_MPEG-2_rdbuf_size;
        functionCode = FUNC_START_PARA;
        MPEG-2VDEC_apply(MPEG-2vdec, in, out);      /* decode sequence header*/
        while (! (decode_out-> end_of_seq) ){     /* not end of sequence  */
             functionCode = FUNC_DECODE_FRAME;
             MPEG-2VDEC_apply(MPEG-2vdec, in, out);  /* decode one picture  */
             decode_out = (DECODE_OUT *)(out[1]);
             if (decode_out-> outputting) {
                 /*********************************************/
                 /* To do -- output the frame                  */
                 /* starting at location decode_out-> outframe */
                 /*********************************************/
             }
             /*************************************************/
```

```
        /* To do -- fill the input buffer between            */
        /* old_ptr and decode_out->next_wptr from source */
        /************************************************/
        old_ptr = decode_out->next_wptr;
    } /* while */
    MPEG-2VDEC_delete(MPEG-2vdec);
    MPEG-2VDEC_exit();
    /* End of program */
```

4.4.5　程序的运行

下面介绍如何创建和运行视频解码器的程序。

1）建立程序

（1）利用 cl6x –@cl.cmd 或者 CCS 自带的工程文件 Mpg2vdec.mak 编译和汇编单个文件。

（2）利用 ar6x @ar.cmd 创建解码器库文件 MPEG-2vdec_ti.lib。

（3）将目标系统与解码器库文件连接起来。

代码的目录里面包含了要编译、汇编和连接到解码器上的所有文件。CCS 的项目文件 Mpg2vdec.mak 能够编译和汇编所有的源文件，并将它们连接到连接器文件 Mpg2vdec.cmd，产生一个独立的可执行文件 Mpg2vdec.out，这个可执行文件能够被下载到 DSP 里，并且能用微软图形用户界面程序进行测试。利用 ar6x 命令，目标文件能够被打包到解码器库文件 MPEG-2vdec_ti.lib 中，而且按照上面的步骤，目标文件还能与目标系统相连。

2）运行程序

（1）运用 CCS 或 evm6xldr 工具下载和启动 DSP 可执行文件 Mpg2vdec.out。

（2）启动微软的图形用户界面程序 Mplay.exe。

（3）在 Mplay.exe 程序界面里，选择“File”，“Open”菜单项，然后选择一个 MPEG-2 格式的视频文件。

（4）在 Mplay.exe 程序界面里，选择“Control”，“Play”菜单项，开始解码。

如果计算机里装有微软的 DirectDraw 软件包，就能看到解码后的视频。

3）检验确认

我们已经用官方提供的 MPEG-2 符合性测试流对解码器进行了全面测试，测试结果证实了此款解码器完全符合要求，这既确保了算法执行的正确性，又保证了输出图像的质量。

4）存储器的需求和性能

下面介绍解码器的内存需求和性能，表 4.6 列出了所有数据和程序对存储器的需求。

表 4.6　解码器的内存需求

	内 存 储 器	外 存 储 器
数据存储器		
头数据存储器	7 552 (7.4 KB)	3 022 848 (2 952 KB)
栈数据存储器	16 384 (16 KB)	
静态数据存储器	10 616 (10.4 KB)	
总计	34 552 (33.8 KB)	3 022 848 (2 952 KB)
程序存储器总计	65 504 (64 KB)	26 432 (26 KB)

外存储器包括输入比特流缓冲器和输出帧缓冲器。它们是由框架分配，分配结果被作为应用编程接口 MPEG-2VDEC_apply 的自变量传递给解码器。

我们以几种 MPEG-2 视频数据流为基准对视频解码器进行了测试，结果如表 4.7 所示。解码器软件是在 C6201 DSP 上运行的，运行时内存储器被设置成映射模式（cache 不起作用）。解码器的性能用 CCS 中运行的标准循环次数来表征。

表 4.7　解码器的性能

测试流	相关信息	格式	比特率（Mb/s）	图片数	性能/MHz	
公共 MPEG-2 流	Mobl_080.m2v.存于网站 from http://www.mpeg.org	704×576 ×25 f/s	7.629	375	平均最 大值	214 239
DVD 测试流#1	Vts_05_1.vob in Panasonic DVD 演示盘	720×480 ×30 f/s	9.346	10 000	平均最 大值	204 255
DVD 测试流#2	Vts_40_1.vob in Philips DVD 演示盘	720×576 ×25 f/s	9.346	1 000	平均最 大值	190 220
MPEG-2 兼容测试流	Bitstream gi_9.m2v in MPEG-2 测试资源库	720×480 ×30 f/s	14.305	16	平均最 大值	261 267

参 考 文 献

［1］ISO/IEC 11172-2, Coding of moving pictures and associated audio for digital storage media at up to about 1.5Mbits/s, Part 2: Video (MPEG-1 video standard).

［2］ISO/IEC 13818-2, Generic coding of moving pictures and associated audio information. Part 2: Video (MPEG-2 video standard).

［3］Texas Instruments, The eXpressDSP Algorithm Standard (xDAIS): Rules and Guidelines. SPRU352.

［4］张旭东，卢国栋，冯健．图像编码基础和小波压缩技术．北京：清华大学出版社，2004.

［5］Ngai-Man Cheung．MPEG-2 Video Decoder: TMS320C62x Implementation．TI ApplicationReport, SPRA649．

第 5 章　TMS320C6416 实现 H.264

H.264 是一个新的、备受关注的视频编码标准。本章给出在 TMS320C6416 处理器上实现 H.264 的一个较详细的说明。通常，在一种 DSP 上实现一种标准算法，包括算法改进和算法实现两个主要部分。算法改进是从算法本身出发，找到更高效率的方法并且保证质量要求；算法实现包括针对选择的处理器，进行各种资源的优化。

本章是视频编码标准在一个 DSP 处理器上实现的综合例子，这个例子很有代表性，它所涉及的不仅仅是单纯的编程技术，很多时候还需要在算法上进行改进。尽管当前人们更多会应用 DM64X，Davinci 来实现 H.264 编解码器，但由于这些处理器的内核均是 TMS320C6416，因而本章的讨论均适用于这些新的媒体处理器的应用。第 6 章是本章的续篇，讨论在 DM642 上实现 H.264 的专题。

5.1　H.264 概述

H.264 是 ITU-T 和 ISO/IEC 联合制定的最新编码标准，它最先由 ITU-T 的 VCEG 于 1997 年提出，目标是提出一种更高性能（相对于当时的 H.263）的视频编码标准，其前身是由 H.263 发展起来的 H.26L，这里 L 代表 Long term，以区别 H.263 版本 2 和版本 3。由于它相对于 MPEG-4 的优良表现，2001 年年底，ISO/IEC 的 MPEG 加入到标准的开发过程中，与 VCEG 组成 JVT。到 2002 年年底，H.264 完成所有技术工作，2003 年年底正式成为官方标准。该标准在 ITU-T 中被称为 H.264，而在 ISO/IEC 中将成为 MPEG-4 的 Part 10 （Advanced Video Coding Profile）。

目前，H.264 标准已经制定完成，从各种实验结果来看，它达到了当初提出的目标。相对于其他标准，H.264 具有以下特点。

1）低码率，高质量

H.264 和其他一些编码标准的比较见图 5.1。从图中可以看出，在相同质量的情况下，H.264 相对于 H.263 的 Baseline 可以节约 40%～50%的码率。

2）广阔的应用范围

H.264 的不同档既可以应用于有严格时延限制的实时通信，也可以应用于对时延要求不高的其他应用（视频存储、流媒体等）。

3）鲁棒性

H.264 在设计时，针对分组交换网（如 Internet）中的分组丢失和无线网络中比特误码都提出了相应的工具，使得 H.264 在这些网络中传播时具有更强的抗误码性能。

Avcrage coding gain for H.26L				
	H.263 CHC	MPEG-4 ASP	MPEG-4 SP	H.263 Baseline
% Bit Savings	24.08	28.34	33.48	42.14
PSNR Gain（dB）	+1.20	+1.41	+1.66	+2.25

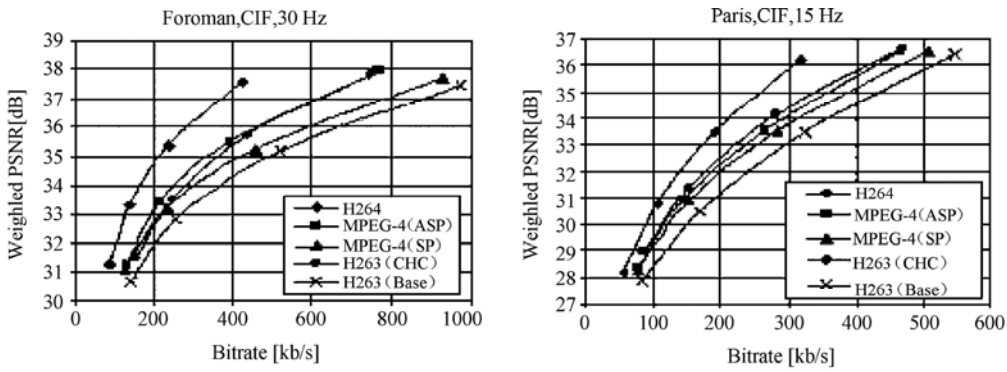


图 5.1　H.264 和其他标准的比较

4）对各种网络的友好性

H.264 中增加了 NAL 层，负责将编码器的输出码流适配到各种类型的网络中，从而提供了友好的网络接口。

和以往视频编码标准相比，H.264 具有突出的性能，它将具有广阔的应用前景和商业价值。它的应用领域包括：

- 通过电缆、卫星、CableModem、DSL 和陆地等媒介的广播。
- 在光学或磁性设备、DVD 上的交互式储存。
- 基于 ISDN、以太网、LAN、DSL、无线移动网络、调制解调器的交互服务。
- 基于 ISDN、CableModem、DSL、LAN、无线网络的视频点播和多媒体流服务。
- 基于 ISDN、DSL、以太网、LAN、无线和移动网络等的多媒体消息服务（MMS）。

5.2　H.264 视频编解码器

和以往视频编码标准类似，H.264 文档中只描述了码流结构与语法，以及实现这些技术的方法，并没有明确规定编解码器是如何实现的，这也给用户开发提供了较大的自由度。一般来说，H.264 编解码器的流程中，除环路滤波器之外，其他的功能模块（预测，变换，量化和熵编码）在以前的标准（MPEG-1，MPEG-2，MPEG-4）也都存在，只是在这些功能模块中采用了比较先进的技术而已。

如图 5.2 所示，编码器包括两条数据流路径，一条前向路径（从左到右），一条为重建路径（从右到左）。解码器的数据流走向从右到左（见图 5.3），其主要模块和编码器类似，只不过是编码器的反过程。在介绍 H.264 的技术细节之前，简单介绍编解码视频中一帧图像的步骤。

- 编码器（前向路径）

F_n 为编码帧（场），F'_{n-1} 为参考帧（场），编码帧是以宏块为单位进行处理的。对于每一个宏块中的当前编码块，都要先得到一个与之对应的预测块，如果是 intra 编码模式，预测块由当前帧当前片中已编码的重建块像素来预测获得，即由图中帧内预测模块完成；如果是 inter 编码模式，通过对前向或后向参考帧进行运动估计搜索，得到当前编码块对应的预测块信息，这部分工作由 ME 和 MC 模块完成。

在得到预测块信息之后，将当前编码块和预测块相减，得到一个残差块 D_n，对其进行变换和系数量化（T 模块和 Q 模块），再对量化后的系数 X 进行熵编码，连同其他被编码的附加信息（预测模式、运动矢量、量化系数等）共同组成压缩码流，到达 NAL（Network Abstraction Layer）进行传输和存储。

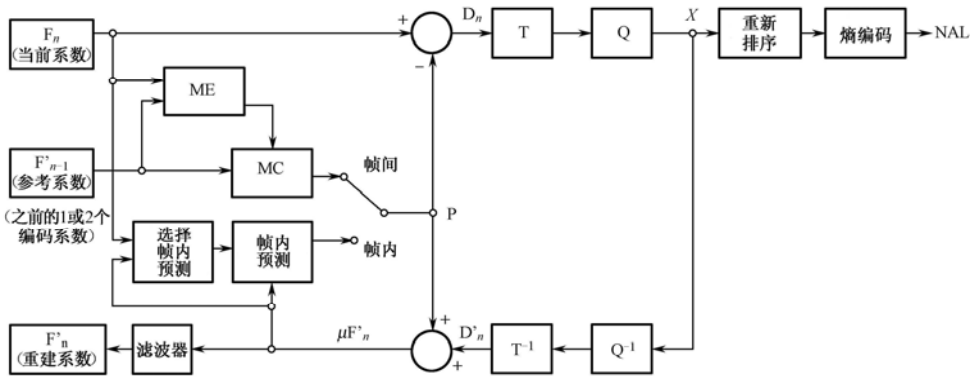


图 5.2　H.264 编码器

➢　编码器（重建路径）

在编码器中，还有一条解码回路，将编码后的数据进行解码，得到重建图像，作为今后编码帧的参考帧。量化系数 X 经过反量化（Q^{-1}）和反变换（T^{-1}），得到残差块 D_n，再把它和预测块相加，得到滤波前的重建块 μF_n，再通过环路滤波器滤波，消除块效应，就得到最后的重建图像 F_n。

➢　解码器

如图 5.3 所示，解码器是编码器的反过程。从 NAL 中得到二进制码流，进行熵解码，得到系数 X，再经过反量化和反变换模块，得到残差块 D'_n，由码流中的头信息，解码器从参考帧中得到预测块数据，二者相加，就得到滤波前的重建数据，再经过去除块效应滤波器，得到最终的解码块信息。

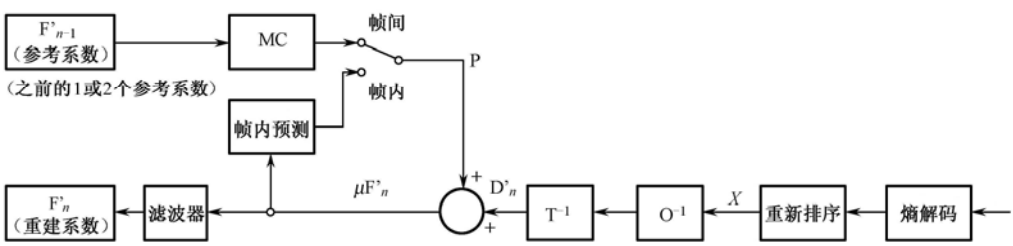


图 5.3　H.264 解码器

5.3　H.264 的结构框架

5.3.1　H.264 的档和层

和其他视频标准一样，H.264 针对不同的应用场合，提供了三种包含不同技术选项的档，分别为 Base_line Profile，Main Profile，Extended Profile（在 2004 年 7 月 JVT Redmond 会议上提出的 H.264/AVC 版本 3 中，又增加了 4 种针对 422、444 格式的档，目前广泛流行的仍然是以上三档）。用户可以根据实际需要，选择不同档的编解码器来实现。不同的档提供了不同的算法要求和限制，使用相同档的解码器，能够解码该档支持的所有特性，而编码器只需支持该档内的部分特性。

1）Base_line Profile

Base_line Profile 应用在视频会议、可视电话等领域，为低码率的压缩算法。它所包含的选项有：

- I 片和 P 片类型；
- 去块效应滤波器；
- 不支持宏块帧场自适应编码；
- Zig-Zag 扫描方式；
- 1/4 像素精度运动估计；
- 三级运动分块，最小块为 4×4 的块；
- CAVLC 熵编码模式；
- 支持任意片顺序（Arbitrary Slice Order）编码，一帧允许有多个片组；
- 支持灵活块顺序（Flexible Macroblock Order）编码方案；
- 4∶2∶0 的色度块采样率；
- 支持冗余片（Redundant Slice）。

2）Main Profile

Main Profile 只应用在广播媒体领域，例如，数字电视，数字广播，视频存储等，为中高码率压缩算法。它所包含的技术特性有：

- B 片；
- CABAC 熵编码；
- 自适应双向预测；
- 除 ASO 编码、FMO 编码和冗余片（Slice）外，支持 Baseline Profile 中的所有特性；
- 支持场编码；
- 支持帧场自适应编码。

可见，Main Profile 除了多片组（Slice Group），ASO，FMO 和冗余片这几个特性，基本上支持 Base_Line Profile 的所有特性。

3）Extended Profile

Extended Profile 在 H.264 早期版本中，又被称为 X-profile。它主要应用于视频流传输等领域。为了支持流媒体在网络上有效传输，Extended Profile 增加了一些差错控制的技术，以提高视频流在恶劣网络环境下的传输性能。它支持的特性有：

- B 片类型；
- SP 和 SI 片类型；
- 数据分层片；
- 自适应双向编码；
- Baseline 中支持的所有特性；
- 支持场编码；
- 支持帧场自适应编码。

H.264 Baseline, Main 和 Extended Profile 的关系如图 5.4 所示。

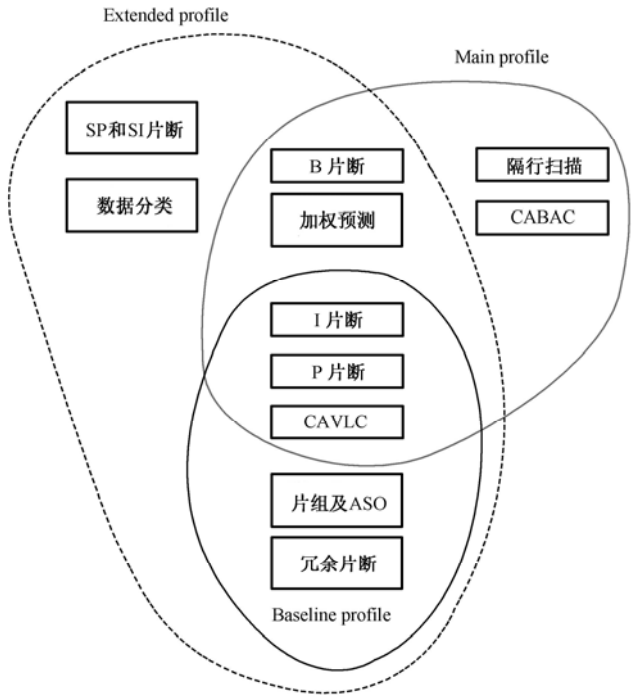


图 5.4　H.264 Baseline, Main 和 Extended Profile 关系图

从以上的介绍可以看出，三档之间关系密切，BaseLine Profile 就是 Extended Profile 的子集，但不是 Main Profile 的子集。它们所采用的技术细节，将在后面的小节中详细介绍。

H.264 标准中，还将档分成若干层，每层都被详细定义了所支持的码率，消耗内存大小，应用领域等。用户可以根据自己的实际需要，以及内存、运算能力的限制，选择合适的层来实现编解码器。表 5.1 列出了不同层的应用范围。

表 5.1　不同层的应用范围

层	应用范围
1.0	QCIF@15f/s
1.1	QCIF@30f/s
1.2	CIF@15f/s
2.0	CIF@30f/s
2.1	HHR@15f/s or 30f/s
2.2	SDTV@15f/s
3.0	SDTV：720×480×20i，720×576×25i 10Mb/s（max）
3.1	1 280×720×30p，SVGA （800×600） 50+p
3.2	1 280×720×60p
4.0	HDTV：1 920×1 080×30i，1 280×720×60p，2k×1k×30p 20 Mb/s（max）
4.1	HDTV：1 920×1 080×30i，1 280×720×60p，2k×1k×30p 50 Mb/s（max）
5.0	SHDTV/D-Cimera：1 920×1 080×60p，2.5k×2k…
5.1	SHDTV/D-Cimera：4 096×2 048

5.3.2　H.264 支持的视频格式

H.264 所支持的原始视频为 YUV420 格式，即每 4 个亮度信号共用一个 C_r 色度信号和一个 C_b 色度信号。在 2004 年 7 月公布的 H.264 新版本中，增加了对 YUV422 格式（每 4 个亮度信号共用 2 个 C_r，C_b 色度信号）和 YUV444 格式（每 4 个亮度信号共用 4 个 C_r，C_b 色度信号）的支持。

此外，场编码中，视频流的一帧被分成奇偶两场，也被称为顶场、底场。其中一帧图像中的奇数行属于奇场，偶数行属于偶场。

5.3.3　H.264 的码流格式

为了使 H.264 码流能够应对不同的应用和网络之间的差异，H.264 中定义了一个 VCL 层（视频编码层 Video Coding Layer）和一个 NAL 层（网络抽象层 Network Abstraction Layer）。VCL 层负责对视频内容进行有效的编码表示，NAL 层负责根据传输层或者存储媒体的特性对码流进行适当的组织。图 5.5 是 H.264 标准的整体框架。

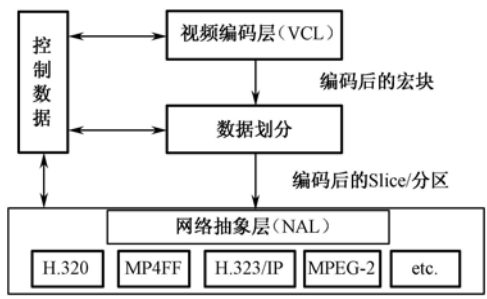


图 5.5　H.264 标准的整体框架

VCL 层编码后的数据，将会被映射成一个个 NAL 单元供传输和存储。每个 NAL 单元包括一个 NAL 头，还有 NAL 的数据包（Raw Byte Sequence Payload，RBSP），这些 NAL 单元就会以数据包的形式在网络中传输和存储。图 5.6 是 NAL 单元序列的结构图。

图 5.6　NAL 单元序列结构图

5.3.4　H.264 的帧结构

视频输入的每一帧图像都被分为若干个宏块，每一个宏块由 1 个 16×16 的亮度块 Y、一个 8×8 的 C_b 块和一个 8×8 的 C_r 块组成。不同的宏块组成不同的块组，称为片（Slice）。H.264 中提出了片组（Slice Group）的概念，片组是由不同的片组成的，是为了灵活块分配编码提出的。

一个视频帧可以编码成一个或多个片，既可以将整帧图像作为一个片进行编码，也可以将每一个宏块作为一个片进行编码。在一帧图像中，每一个片所包含的宏块数可以不一样，两个片之间编码独立，不具有相关性，这样有助于降低差错传播，增强编码的鲁棒性。每一帧中的片可以具有不同的编码模式，例如，在一帧 Baseline Profile 的图像中，可以有 I 模式的片，也有 P 模式的片。

H.264 中允许采用片组进行分片，所谓片组，就是把一帧的宏块划分开，分成不同的部分，每一部分属于一个片组，然后属于一个片组内的宏块组成不同的片。H.264 中定义了 7 种片组类型，例如，交织类型片组，属于不同片组的宏块在图像中所处的位置相互交织在一起。图 5.7 是前景背景型的片组，这样的分块结构使得编码更加灵活。例如，视频电话时，一般都是背景不变化，而前景在变化，而且一般前景都是在中央，就可以把背景的宏块定义成一个片组，把前景的宏块定义成另一个片组，这样在同一个片中的数据具有更大的相关性，在编码时，会进一步降低码率。

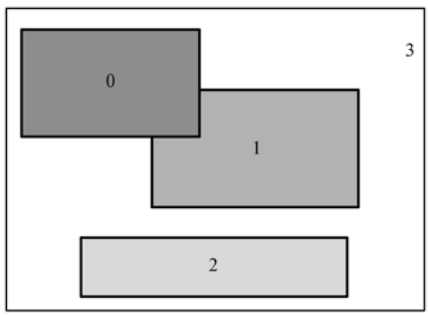


图 5.7　前景背景型片组结构的示例

宏块（16×16）的下一级是块（8×8），每个宏块包括 4 个 8×8 的亮度块，以及一个 8×8 的 C_r 块和一个 8×8 的 C_b 块，其中每块又分成 4 个 4×4 的子块。

H.264 的总体帧结构如图 5. 8 所示。

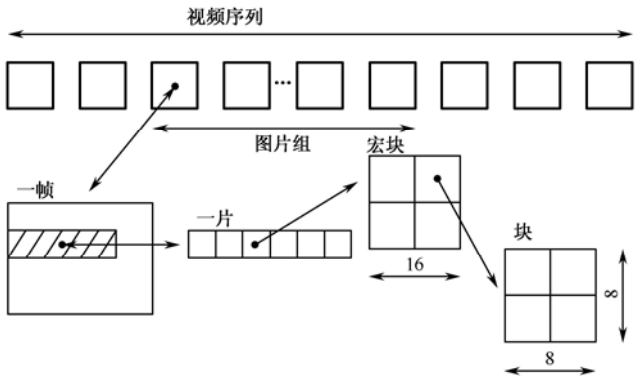


图 5.8　H.264 的总体帧结构图

5.4　H.264 具体技术概述

H.264 协议的主要目的是提供一种和现有视频编码标准相比具有更高编码质量的视频压缩标准，但 H.264 中主要的方法和以前的标准（如 H.263、MPEG-4）类似，都包括以下四个阶段。

- 将视频帧分成块，以便将帧的处理在块的层次上进行。
- 利用视频帧内存在的空间冗余性，将一些原始的块进行变换、量化和熵编码处理。
- 利用连续的帧之间具有的时间冗余性，通过运动估计和运动补偿，仅仅对帧之间变化的部分进行编码。对于一个给定的块，在前一个或者多个编码帧中搜索确定运动矢量，以便编码器和解码器用来进行该块的估计。
- 利用帧内的统计冗余性对块的残差信息进行变换、量化和熵编码处理。

H.264 的主要编码流程与先前的一些编码标准（如 MPEG-1, MPEG-2, H.263）相比并没有结构性的变化，而是在各个主要模块内部使用了一些先进的技术，提高了编码效率，这些新技术包括：

✧ 帧内预测编码；

✧ 新的运动估计方法。

- 可变块大小；
- 多帧运动估计；
- 1/4 甚至 1/8 像素精确度运动估计；
- 解码环路中的去块滤波器。

✧ 整数 DCT 变换；

✧ 新的熵编码方法。

- CAVLC（Context-based Adaptive Variable Length Coding）；
- CABAC（Context-based Adaptive Binary Arithmetic Coding）。

以下将对这些新技术进行简单介绍。

5.4.1　帧内预测编码

在视频编码中，通常的方法是把整幅图像分为若干部分（宏块），然后对每一部分进行编码，在编码时采用 Intra 或 Inter 两种模式。Inter 模式采用运动估计技术，对预测残差进行编码，其中使用的技术将在后面进行介绍。传统的 Intra 技术则直接对原始像素进行编码。可以看到，Intra 模式没有像 Inter 模式那样利用视频序列中的时间冗余度，因而会产生比较高的码率，但好处是每个宏块单独编码，可以防止误码的扩散及支持随机访问，等等。

在 Intra 模式中，通常直接对宏块进行 DCT 变换，对变换系数进行熵编码，这样做虽然一定程度上消除了帧内的空间冗余度，但是由于 DCT 只是利用了宏块内部像素之间的相关性而没有考虑到相邻宏块间的相关性，因而，传统的 Intra 编码对视频序列空间冗余度的利用也是不完全的。在 H.264 中，为了提高 Intra 模式的编码效率，引入了 Intra Prediction 的方法，它使用相邻的宏块对编码宏块进行预测，对预测残差进行变换编码。针对不同的块大小（16×16，4×4），不同的块类型（亮度，色度），H.264 都规定了一系列的预测方法。

对于亮度块，可以对 16×16，4×4 两种块大小类型进行预测。其中 4×4 块大小的 Intra 预测模式有 9 种，分别为垂直预测，水平预测，直流（DC）预测，DDL（Diagonal Down-left）预测，DDR（Diagonal Down-Right）预测，VR（Vertical Right）预测，HD（Horizontal-Down）预测，VL（Vertical Left）预测，HU（Horizontal Up）预测。图 5.9 给出了 4×4 块预测的像素，图 5.10 表示各个模式的预测图样。除了模式 2 是直流预测模式，其他都是方向预测模式。

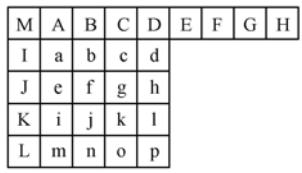


图 5.9　4×4 Intra 预测使用的像素

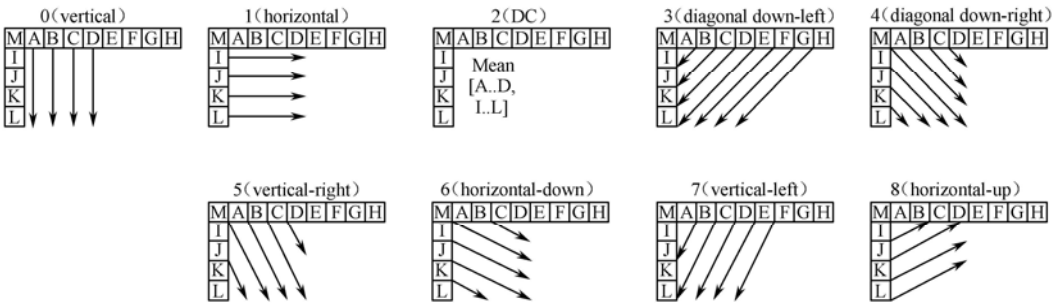


图 5.10　4×4 Intra 预测模式

如图 5.11 所示，对于 16×16 大小的亮度块，标准定义了 4 种预测模式，分别为水平预测、垂直预测、直流预测及平面预测。在预测过程中，使用的像素值分别位于当前块上一行、左边一行及左上角相邻顶点，共 33 个点。

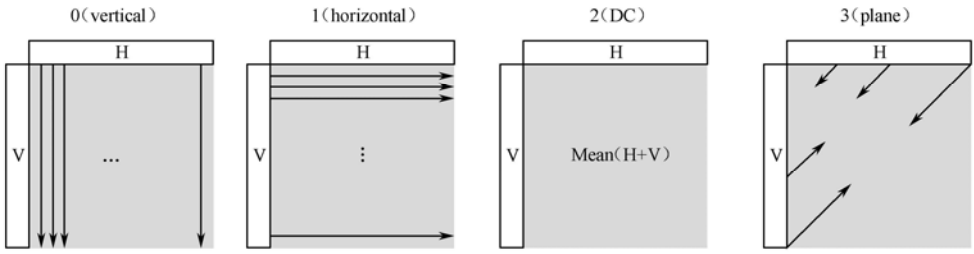


图 5.11　16×16 Intra 预测模式

对于色度块，标准规定对 8×8 块大小进行 Intra 预测，共有 4 种模式，与 16×16 Intra 预测非常类似，也是分别为垂直、水平、直流、平面预测，只是具体算法略有不同。

需要说明的是，在 JPEG-2000，H.263+，MPEG-4 中也提出了类似的帧内预测方法，但不同之处在于 H.264 是直接在空域中进行预测，而其他标准则是在变换域中进行预测。

5.4.2　运动估计

在视频中，图像是由连续的帧组成的图像序列，由于景物变化速度的限制，相邻帧之间存在很高的相关性，即存在很高的时间和空间的冗余。如何消除这些冗余就成为视频编码中的关键技术，运动估计技术的目的就是消除时域冗余，利用已编码参考帧的信息，对当前编码帧进行预测，对差分信号进行变换编码，能得到很高的压缩比。由运动估计补偿技术，结合变换编码，构成了视频编码的主要方法。

运动估计是应用于帧间编码的一项技术，和帧内编码的 I 帧对应，帧间编码有两种帧格式，分别为 P 帧和 B 帧，其中 P 帧是前向预测，也是单向预测，即利用序号位于当前编码帧之前的已编码帧作为参考帧，对当前帧进行预测。B 帧是双向预测，即利用当前帧前向、后向的已编码帧作为参考帧，对当前帧进行预测，图 5.12 表示了 3 种帧结构的相互关系。

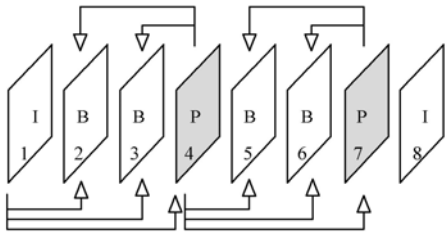


图 5.12　3 种帧结构的相互关系示意图

H.264 和以往的视频编码标准相比，在运动估计方面采用了许多新的技术，主要包括可变块大小，多帧运动估计，亚像素的运动估计及 de-blocking filter 等。

1）可变块大小

进行运动估计时使用的块大小对运动估计的效果有较大的影响。在 H.263 和 MPEG-4 中已经提出过可变块大小的概念，但 H.264 中提供了更多的块大小供选择。在标准中，每一个亮度宏块（16×16）可以被分成 4 种块大小，分别为 16×16，8×16，16×8，8×8。在运动

估计过程中，如果 8×8 块大小模式被选中，则可以进一步分成 8×4，4×8，4×4 的块大小模式。如图 5.13 所示，共有 8 种不同的块大小类型，这使得一个宏块内部可能包含多种块大小模式。

编码过程中，每一个子块都有独立的运动矢量，这些运动矢量连同块大小类型都将被编码，成为码流的一部分。选择较大的块类型（16×16，8×16，16×8）意味着运动矢量将占用较少的码流，但是运动估计将比较粗糙，残差场的能量也将比较高，会耗费较多码率。如果选择较小的块类型（8×8，8×4，4×8，4×4），运动估计比较精细，每块的残差场能量很低，但是运动矢量和块类型信息将占用较多码字。块大小的选择对最终编码结果有较大影响，一般来说，运动比较平缓的区域，如静止背景等，适合用较大尺寸的块进行预测，运动比较剧烈和具有较多细节的区域，适合用尺寸较小的块进行运动估计。

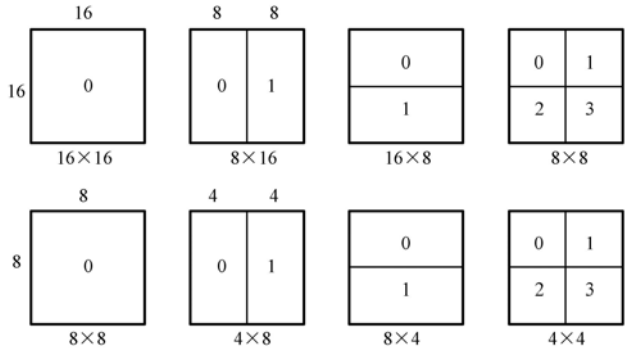


图 5.13　8 种可变块大小

对于色度块，由于尺寸是亮度块的一半，它的块模式选择与对应的亮度块一致，只是尺寸减半。例如，16×8 的亮度块所对应色度块的尺寸为 8×4，8×4 的亮度块对应的色度块大小为 4×2。同理，色度块的运动矢量的分量，也为对应亮度块的运动矢量各分量的一半。

2）亚像素运动估计

在 H.26l 中采用整像素运动估计，在 MPEG-4 和 H.263 中，采用了半像素精度的运动估计算法。在 H.264 中，更是把运动估计的精度提高到了 1/4 像素，并且把 1/8 像素精度的运动估计作为一个可选项。H.264 中使用 1/4 像素运动估计，和整像素相比可以节省 20%的码率。

和半像素精度的运动估计一样，1/4 像素精度的运动估计使用插值滤波算法，得到半像素和 1/4 像素位置的点。如果运动矢量指向整像素位置，那么预测数据由相应的参考图像的像素组成；否则相应的非整位置像素值需要用插值方法获得。在 H.264 中，采用一个六阶滤波器插值得到亚像素点的值。

图 5.14 中，灰颜色的点代表整数位置的像素点，其他的为小数位置的像素点。一个典型的内插过程如下所述。

首先，由整数位置内插得到半像素位置。

```
b1=E-5F+20G+20H-5I+J
h1=A-5C+20G+20M-5R+T
j1=cc – 5*dd + 20*h1 + 20*m1 – 5*ee + ff 或者
```

```
j1=aa – 5*bb + 20*b1 + 20*s1 – 5*gg + hh
b=Clip1((b+16)>>5)
h=Clip1((h+16)>>5)
j=Clip1((j1+512)>>10)
```

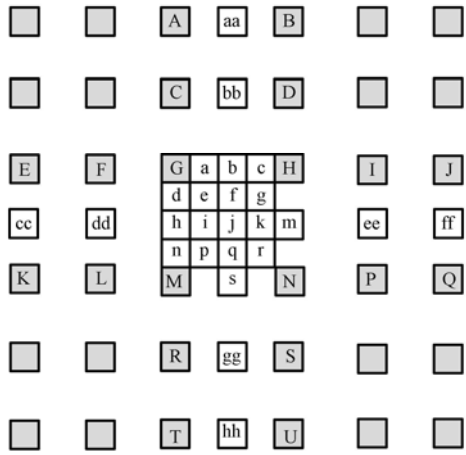


图 5.14　1/4 像素精度的运动估计

其中 aa，bb，gg 等都是诸如 b1，h1 的未被 Clip1 函数滤波前的半像素点值，Clip1(x)为饱和函数，功能是将像素值限制在 0～255 之间。

对于 a，c，d，n，f，i，k，q，由最近的半像素点和整像素点的向上进位的平均得到，例如：

```
a=(G+b+1)>>1, c=(H+b+1)>>1
```

对于 e，g，p，r，采用对角线方向的进位平均得到，例如：

```
e=(b+h+1)>>1, g=(b+m+1)>>1
```

由于色度块运动矢量是亮度的一半，因此色度具有 1/8 精度的运动矢量。色度块采用线性插值的方法，每一个 1/8 精度的亚像素值，由周边 4 个整像素值的线性组合得到。图 5.15 是色度插值的示意图。图中，*A*，*B*，*C*，*D* 为整像素位置，dx，dy 为色度运动矢量的小数部分，取值为 0～7，插值公式如下：

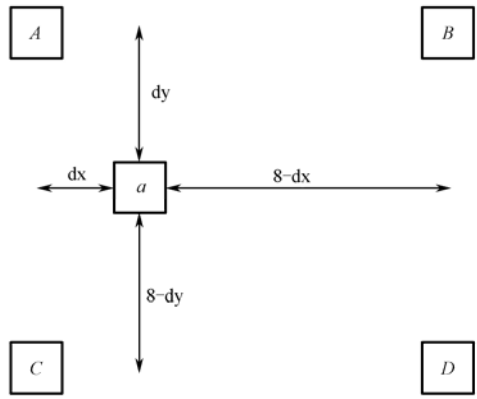


图 5.15　色度插值示意图

a=round([(8−dx)(8−dy)A+ dx(8−dy)B+(8−dx) dy$C$+dxdy$D$]/64)

3）多帧运动估计

事实上，多帧运动估计指的是运动估计中的一大类方法，它们的共性是使用不止一个参考帧来估计运动矢量。以往的视频编码标准只使用至多两个参考帧进行运动估计（B 帧编码），而 H.264 允许前向后向的参考帧可以从 1～15 帧不等。如果把以前的视频编码标准中使用的单帧运动估计技术视为多帧运动估计技术的一个特例的话，那么，多帧运动估计的优点是显而易见的。

根据帧缓存中参考帧的类型，H.264 使用的多帧参考技术有两种：长期记忆运动估计和短时参考运动估计。前者是将一些关键帧长期放在帧缓存中，作为参考帧；后者是将当前编码帧前后的已编码帧放入帧缓存，作为短期参考帧，短期参考帧会由滑动窗口协议自动更替。运动估计时，在多参考帧中寻找预测误差最小的运动矢量，这也是对单帧运动估计的简单扩展。多帧运动估计示意如图 5.16 所示。

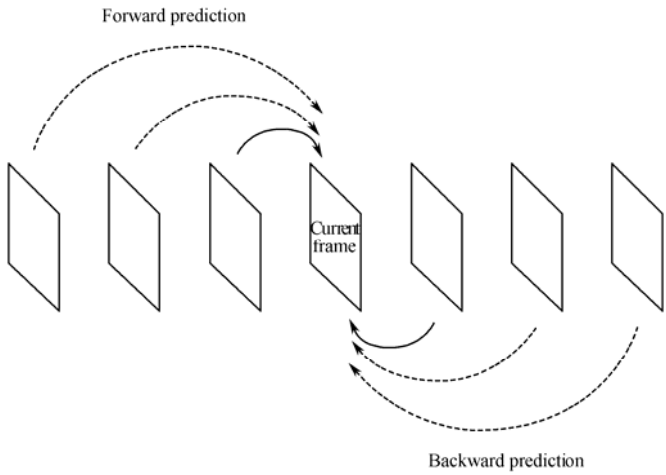


图 5.16　多帧运动估计示意图

多帧运动估计相对于普通的单帧预测具有下述优点：

- 运动估计将更有效率。如果把运动估计视为一个矢量量化的问题，那么，随着参考帧的增多，意味着量化码书的变长，从而允许进行更精确的量化。此外多参考帧在小运动序列及“振动序列”（如摇头动作）等环境中非常有用。
- 更强的差错鲁棒性。由于帧间编码使用了前面的帧作为参考，所以一旦一帧中出现了错误，将会影响到后面的帧，从而导致错误的传播。如果在解码器和编码器之间存在一个反馈回路，那么解码器就可以通知编码器发生错误的帧，强迫编码器使用其他帧进行运动估计，从而防止差错的传播。

当然，多帧运动估计也有不足之处，比如内存需求的增大和运算复杂度的增加，这些都是需要研究和克服的方面。

4）去块效应（环路）滤波器

实际上 De-blocking Filter 在 H.263 的 Annex J 中已经被提出，但是并没有要求一定使用。De-blocking Filter 的作用就是用来消除解码图像中的块效应。块效应产生的原因是由于各个宏块分别进行量化，这样在相邻宏块的交界处，由于量化步长不同导致原本取值很接近的像素重构后产生了较大的差异，形成明显的块边界。H264 中的 De-blocking Filter 通过在 4×4 的块边界上进行滤波，使块边界趋于缓和，从而达到去除块效应的目的。

环路滤波器作用于编码器和解码器的运动补偿之后，重建帧之前，标准中对滤波器的使用有详细的说明，在每一个 4×4 块的边界处，通过计算边界强度值，对边界配置不同的滤波器。此外，标准中还定义了两个阈值 α、β，它们由量化参数决定，量化参数越大，则边界处块效应也越大，α、β 也随之增加。如果边界两边像素值之差在阈值范围之内，则需要进行滤波。若边界像素值相差高于阈值，则认为该处为图像固有边界，不进行滤波。图 5.17 为量化参数 36 时，有无环路滤波器作用的效果对比，可以看出，环路滤波器去除块效应效果非常显著。

（a）无环路滤波　　（b）有环路滤波

图 5.17　环路滤波器作用的效果对比

5.4.3　整数 DCT 变换

由于 DCT 的性能十分接近统计意义上的最优变换 KLT，而且具有快速算法，所以 DCT 被广泛地应用于各种视频编码标准中，但是，传统的 DCT 无论在运算精度还是复杂度上都存在明显的不足。

在运算精度方面，由于变换中存在无理数，这样在变换时不得不对变换后的系数进行四舍五入，从而导致反变换后不能精确恢复原始数据。而且由于 DCT 存在多种快速算法，当编码器和解码器适用的算法不能很好匹配时，就使得编码器－解码回路的解码结果（用做编码器的参考帧）和解码器的解码结果（用做解码器的参考帧）不一致，产生了参考帧的偏移，这就是传统 DCT 中常见的不匹配问题，严重时对重建图像的质量会有很大的影响。

在复杂度方面，由于变换中多次使用乘法，导致 DCT 运算十分耗时，例如，在 MPEG-1 中，统计表明，IDCT 占解码负载的 22.10%，仅次于运动补偿（38.64%）。

由于传统 DCT 的这些弱点，在 H.264 中引入了整数 DCT 变换，它的正反变换矩阵分别为：

$$\boldsymbol{H}=\begin{bmatrix}1 & 1 & 1 & 1\\ 2 & 1 & -1 & -2\\ 1 & -1 & -1 & 1\\ 1 & -2 & 2 & -1\end{bmatrix} \qquad \tilde{\boldsymbol{H}}_{\text{inv}}=\begin{bmatrix}1 & 1 & 1 & \frac{1}{2}\\ 1 & \frac{1}{2} & -1 & -1\\ 1 & -\frac{1}{2} & -1 & 1\\ 1 & -1 & 1 & -\frac{1}{2}\end{bmatrix}$$

注意其中的系数基本上都是整数（1/2 可以用移位代替），这样不但解决了精度问题，而且由于乘法均可由移位运算代替，运算的复杂度也大大降低。

需要注意的是，$\boldsymbol{H}$ 和 $\tilde{\boldsymbol{H}}_{\text{inv}}$ 并不是严格意义上的互为逆矩阵，确切的讲，$\tilde{\boldsymbol{H}}_{\text{inv}}$ 应该是 $\boldsymbol{H}$ 的按比例反，依次进行正反变换的时候，相当于原始像素矩阵每一个点都被乘以不同的因子。不过，这一点对重建没有影响，可以通过合理设计量化表来进行补偿。

H.264 中的宏块大小为 16×16，对其中每个 4×4 大小的块进行上述 4×4 的 DCT 变换后，得到 16 个 4×4 变换矩阵。为了进一步提高压缩效率，H.264 还允许把每个 4×4 变换矩阵中的直流分量（位于矩阵左上角的元素）单独取出，组成一新的 4×4 矩阵，对此矩阵进行 Hardamard 变换。

利用上述矩阵变换后，它的逆变换可以精确地用整数表示，因此逆变换误差可以消除。变换编码的过程和以前的标准很相似，编码器包括正变换、Zig-Zag 扫描、按比例缩放、量化和熵编码，而解码器将上面的顺序倒过来就可以了。

5.4.4　熵编码

在 H.264 中，有两种熵编码方法：CAVLC（Context-Adaptive Variable Length Coding）和 CABAC（Context-based Adaptive Binary Arithmetic Coding）。

当选择 CAVLC 编码模式时，变换系数使用 CAVLC 进行熵编码，其核心仍然是变长编码，只不过可以随着编码的进行，根据周边邻块信息自动选择不同的码表，能更充分地与信源的统计特性相匹配。其他的码流元素，如运动矢量、量化参数等采用 Exp-Golomb 码的变长编码方法。

H.264 中用到的另外一种熵编码方式就是 CABAC。相对于 CAVLC，CABAC 的编码效率更高，这是因为它采用了算术编码的方法，不受 1 比特/符号的限制，即一个符号可以用少于 1 比特来表示。CABAC 的主要编码流程如图 5.18 所示。

上下文模型的作用是根据上下文情况为待编码的符号选择一个合适的上下文模型，每个上下文模型可以视为一个自动状态机，它的状态是一个二元组（MPS，PLPS）。MPS 英文为 Most Probable Symbol，代表 0 和 1 中概率较大者（注意到这里是二进制算术编码，所以编码的对象为 0、1 组成的二进制序列），而 PLPS 代表 LPS 的概率。每当编码引擎编完 1 比特，它都根据情况更新上下文模型的状态，这也就是图中概率估计模块的作用。

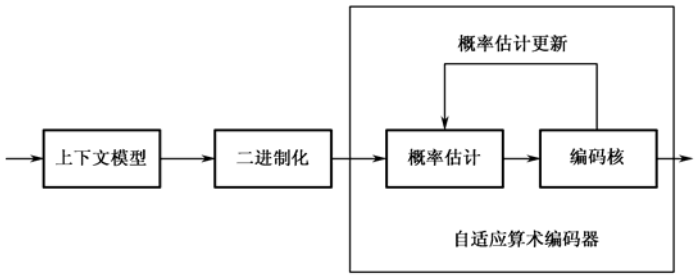


图 5.18　CABAC 编码流程

H.264 中针对各种不同的语法元素（如宏块类型、运动适量残差等）定义了多达 227 种的上下文模型。

CAVLC 和 CABAC 相比较，两者各有所长，从编码效率来看，CABAC 明显要强于 CAVLC，图 5.19 为根据实验所得数据绘制的 R-D 曲线。

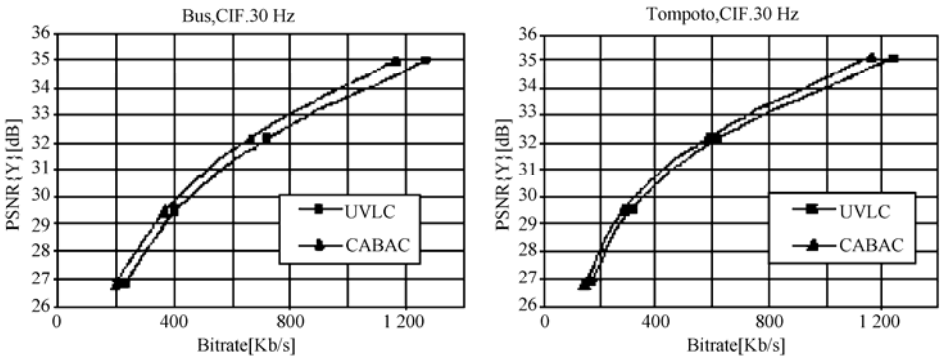


图 5.19　R-D 曲线

从图中可见，在所有码率下，CABAC 的表现都强于 CAVLC。但是需要注意的是该实验是在无误码的假设下进行的，而实际中的信道（如 Internet 或是无线信道），都不可避免有误码的产生，在这种情况下，CAVLC 的抗误码性能要强于 CABAC，而且，从运算的复杂度来看，CAVLC 也远远低于 CABAC，正因为如此，H.264 规定在 Baseline Profile 中使用 CAVLC，而在 Main Profile 中采用了 CABAC 进行熵编码。

由以上介绍和结果可见，H.264 的大部分编码增益来源于其更有效的运动估计和熵编码技术，实验数据表明：

- 可变块大小可以降低大约 16%的码率；
- 1/4 像素精度运动估计可以降低约 20%的码率；
- 多帧运动估计可以降低约 5%～10%的码率；
- 使用 CABAC 作为熵编码可以降低约 10%的码率。

整数变换的使用，使得变换的精度和运算复杂度都得到了很大的改善。解码回路中的 De-Blocking Filter 缓解了块效应的影响，同时使得运动估计的准确性得到了进一步保障。

除了以上提及的一些主要技术之外，H.264 中还使用了一些以下的其他技术：

- 双向预测帧（B 帧）；
- 新定义的 SP/SI 帧（用于视频序列间的切换）；
- 加权预测（Weighted Prediction）；
- 灵活的宏块顺序 FMO（Flexible Macroblock Ordering）；
- 任意片顺序 ASO（Arbitrary Slice Ordering）；
- 数据分区（Data Partition）。

具体的技术细节可参见 H.264 标准的正式文档，这里就不赘述了。

5.5　实现 H.264 编解码的 TMS320C6416 平台

前几节介绍了 H.264 标准的技术特点，这一节，将介绍一个在 TMS320C6416 上实现 H.264 的设计实例，选择 TMS320C6416 的一个典型开发平台 NVDK（Network Video Development Kit），在此平台上实现一个 H.264 的编码器。

NVDK 板的核心芯片是一款 TMS320C6416（简称 C6416）。C64 系列 DSP 是目前 TI 公司推出的处理能力最强大的通用定点 DSP，其最高主频目前已经能达到 1.1 GHz，加上每个指令周期能够最多同时处理 8 条指令，其运算能力可达 8 800 MIPS。目前 TI 公司推出的非常流行的专用媒体处理芯片 DM642，其核心也是一个 C64 内核。这里使用的 C6416DSP，最高主频 600 MHz，是处理能力最强的一款定点 DSP。下面简要介绍 C6416 的有关技术特点。

5.5.1　TMS320C6416 简介

TMS320C6416 是高性能的通用定点 DSP，它基于 TI 公司推出的第二代高性能，先进的 VelociTI^{TM} 超长指令字（VLIW）结构（VelociTI.2^{TM}），使其成为多路、多功能应用领域的出色选择。此外，它还与 C6000 TM 系列 DSP 的代码完全兼容，方便代码移植。图 5.20 是 TMS320C6416 的体系结构框图。

这款 DSP 最高主频 600 MHz，运算能力 4 800 MIPS，对高性能计算提供强大的处理能力。C6416 具有操作灵活的高速 DMA 控制器和强大的阵列处理能力。它的核心处理器具有 64 个 32 位通用寄存器，以及 8 个独立功能单元——2 个乘法器和 6 个算术逻辑单元（ALU），并具有 VelociTI.2^{TM} 的扩展功能。扩展功能包括能够加速运算性能及增强并行性的新指令。它能够在一个时钟周期处理 4 个 16 位乘法运算（MACs），每秒达到 2 400 MMACs。或者单时钟周期处理 8 个 8 位乘法运算，每秒 4 800 MMACs，此外 C6416 还具有专用硬件逻辑，片上内存及丰富的片上外设。

C6416 还具有两个高性能的嵌入式协处理器：维特比译码协处理器（VCP Viterbi Decoder Coprocessor）和 Turbo 译码协处理器（TCP Turbo Decoder Coprocessor），这将增强芯片的信道译码能力。VCP 工作在 CPU 时钟 4 分频下，TCP 工作在 2 分频下，它们之间的通信及和 CPU 的通信都通过 EDMA 控制器来完成。

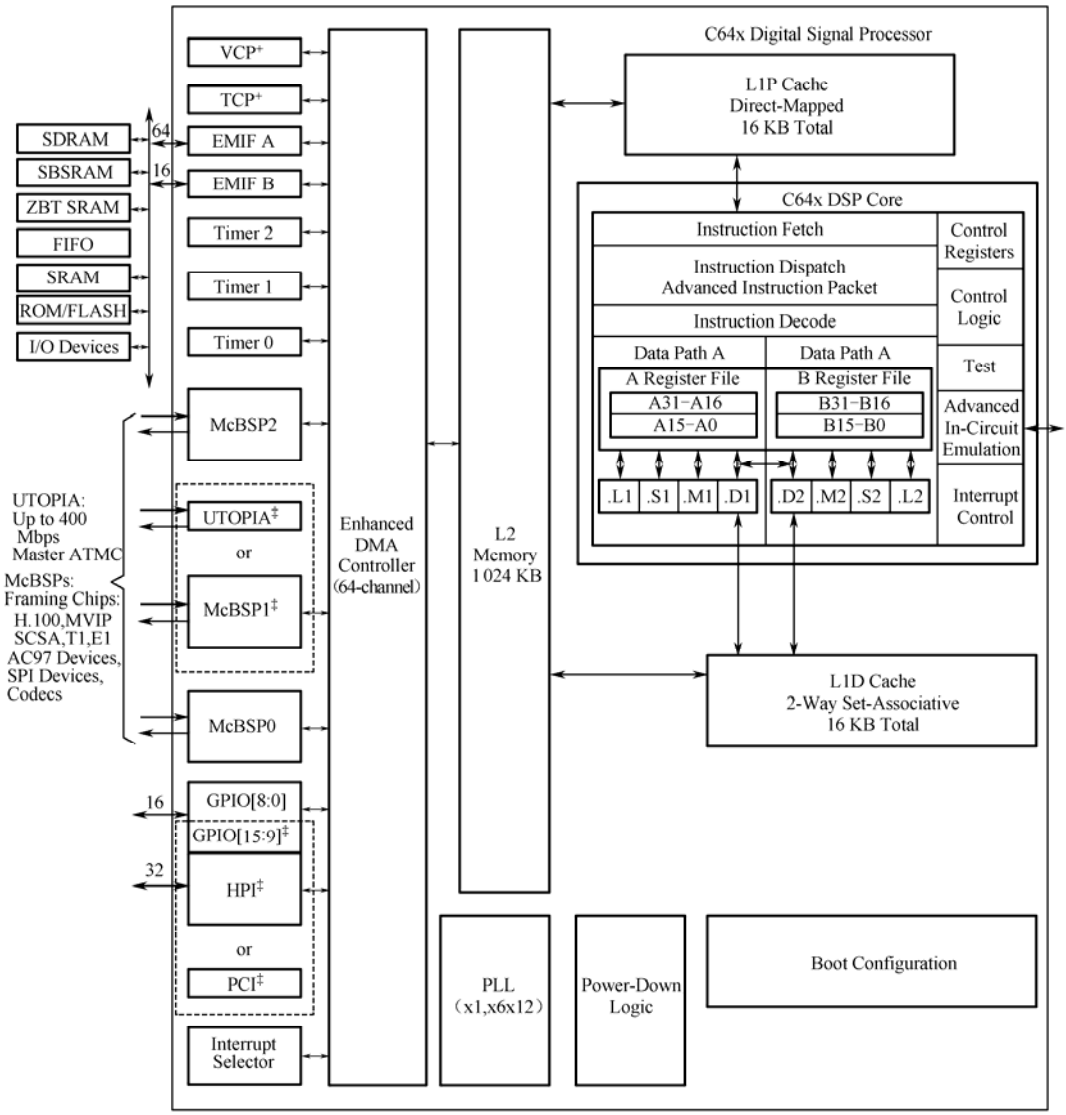


图 5.20　TMS320C6416 的体系结构框图

C6416 具有两级 Cache 结构及功能强大的各种外设。一级程序 Cache（L1P）和一级数据 Cache（L1D）大小均为 16 KB。片上 SRAM 共有 1MB，为数据和程序共享空间，可以通过配置，将其一部分空间作为二级 Cache 使用，最大可划分 256 KB（2Mb）的空间。外设包括三个功能强大的多通道缓存串口（McBSPs）；三个 32 位通用定时器；一个用户可配置的 16 位/32 位宿主机接口；一个 PCI 总线接口；一个 16 脚通用输入输出接口（GPIO）；两个无缝连接的外存接口，分别为 64 位宽度的 EMIFA 和 16 位宽度的 EMIFB，它们都能够和同步（SRAM、EPROM）或异步（SDRAM、SBSRAM、FIFO）存储器相连，总共具有 1280MB 的可寻址扩展内存空间）；此外，C6416 还具有 64 个 EDMA（Enhanced Direct-Memory-Access）独立通道，可以使内存访问相对于 CPU 后台操作，不占用 CPU 资源，大大提高存取速度。

由于 C6416 的强大处理能力和丰富的外设，用户可以灵活地对其进行开发与应用，被广

泛应用于媒体处理、信号检测、数字通信等多通道、多用途的应用领域。

5.5.2　CPU 的技术特点

C6416 的 CPU 内核包括以下组成部分：

- 2 个通用寄存器组（A 和 B），各有 32 个，共 64 个 32 位通用寄存器；
- 8 个功能单元，包括 2 个乘法单元（.M1 和.M2），6 个算术逻辑单元（.L1、.L2、.S1、.S2、.D1、.D2）；
- 2 个读内存的数据通道（LD1 和 LD2）；
- 2 个写内存的数据通道（ST1 和 ST2）；
- 2 个数据寻址通道（DA1 和 DA2）；
- 2 个寄存器组间数据交叉通道（1X 和 2X）。

图 5.21 是 C6416 CPU 的结构框图。

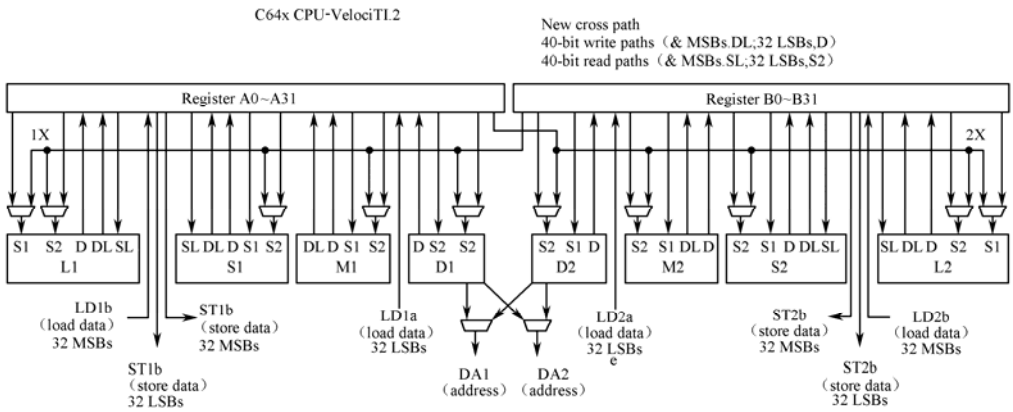


图 5.21　C6416 CPU 结构框图

C6416 的 CPU 采用超长指令字（VLIW）结构，一个时钟周期能同时执行 8 条 32 位指令。它将指令合并成一个最大 256 位宽的指令包。每条 32 位指令的首位决定下一条指令是否属于当前时钟周期执行的指令包。此外，执行指令包也并不总是 256 位，它是变长的，根据当前时钟周期同时执行的指令个数变化，这样有效节约了缓存，这也是 C64 的 CPU 区别于其他微处理器超长指令字结构的显著特征。和 TI 上一代 DSP 产品 C62 相比，C64 CPU 的功能有了显著增强，这些特点包括：

1）寄存器组的增强

寄存器的个数增大一倍，C62x 有 32 个通用寄存器，而 C64x 有 64 个通用寄存器。

C62x 使用 A1、A2、B0、B1、B2 作为条件寄存器，而 C64x 将 A0 也作为条件寄存器，使其总数达到 6 个。

C62x 寄存器支持紧缩 16 位数据、32 位数据和 40 位数据。C64x 在此基础上还支持紧缩 8 位数据（即一个寄存器存放 4 个单字节数据），64 位数据。

2）数据通道的扩展

每个.D 单元能够在单指令中读写双字数据（64 位），而 C62x 的.D 单元在一条指令内无法实现这样的功能。

.D 单元也和.L、.M、.S 单元一样，能通过数据交叉通道访问数据，而 C62x 中的.D 单元只支持寻址交叉通道。

C64x 支持流水线结构交叉数据访问，允许相同寄存器作为交叉数据通道立即数在一个指令包内被多个功能单元访问，而在 C62x 中，每个时钟周期，同时只有一个功能单元能够通过交叉数据通道从相应的寄存器组中获取数据。

3）紧缩数据处理

针对紧缩数据的指令集增加，大大增加了指令的并行性。例如，增加了同时处理 4 个 8 位数或 2 个 16 位数的指令。

扩展的紧缩和非紧缩指令集简化了紧缩数据类型的处理。

4）增强的功能单元硬件

每个乘法单元一个时钟周期可以执行 2 个 16 位×16 位乘法运算，或者 4 个 8 位×8 位运算。

.D 单元通过 non-aligned 读写指令，能够访问任意字节边界的字或双字数据。C62x 只提供 aligned 读写指令，只能读写对齐的数据，很不灵活。

.L 单元能同时执行 4 对 8 位数求差绝对值的运算，这对于视频处理中运动估计运算非常有用。

.L 单元能执行字节移位，.M 单元能执行双向任意位移位，这作为.S 单元移位功能的补充，对音频压缩方面的应用很有帮助。

.M 单元还增加了一些特殊的通信专用指令，如 SHFL，DEAL 和 GMPY4 等。

.M 单元扩展了位统计，位交换一类的指令，这极大地支持位级的算法应用，例如，加密算法等。

5）增强的指令集正交性

除.S 和.L 单元外，.D 单元也能执行 32 位的逻辑指令，.D 单元支持直接读写双字数据的指令，而 C62x 不能直接读写双字数据，C67x 只支持读双字数据。

除了.S 单元能加载 16 位常数外，.L 和.D 单元也能用来加载 5 位的常数。

C6416 超越了硬件上的扩展，在数字数据处理领域，具有高水平的性能。CPU 结构的紧密耦合及出色的编译器使它有最大的处理能力。类似 RISC 结构的指令集及流水线应用的扩展，使得许多指令具有高度的并行性，这是其具有出色性能的关键原因。此外，高性能的两级缓存结构设计，使得 CPU 能够运行在最高的速率。两级缓存通过片上片外数据的自动交换，降低了程序的运行时间。高性能的 EDMA 控制器也使 CPU 具有高效率。

受益于先进的紧缩指令集、数量加倍的通用寄存器及数据线宽度，C6416 的编译器能够基本不受系统结构的限制，使代码产生很高的性能。这也使得 C64x 系列 DSP 成为 TI 先进的数字信号处理器，能够满足不断发展的应用需求。

5.5.3　NVDK（Network Video Development Kit）简介

TI 的网络视频开发套件 （NVDK）是一个用于多媒体系统开发的工具。该套件主要针对高性能的可编程数字信号处理器 TMS320C64X™ 平台的应用，其核心是一块 TMS320C6416 芯片，可实现数字媒体应用的快速开发。该套件为诸如视频基础设施及网络化视频设备等高级视频应用制造商提供了方便，使他们在数字视频解决方案开发进程中得到所需的主要软/硬件资源。

由 TI 第三方 ATEME 开发的 NVDK 套件包括：基于 TI 公司 TMS320C6416 的 ATEME 视频评估板、10/100 Mb/s 的以太网子卡、音频/视频接口盒及电源。还包括带有诸如原理图、驱动程序、板级支持库及应用源程序示例等主要软件与文档光盘。

NVDK 板还采用了多种网络接口，以满足日益增长的连接性需求。TI 的传输控制协议/因特网协议（TCP/IP）栈可在 C6000 上运行，从而使之在没有网络处理器及相关软件的情况下也可连通网络，降低了总体系统成本。TCP/IP 栈软件具有足够的性能空间、灵活性及易于集成且符合 API 等特点。

1. NVDK 体系结构及技术特点

NVDK 板基于 TI 公司的 TMS320C64 系列 DSP，板上 DSP 为 TMS320C6415 或 TMS320C6416，工作频率为 600 MHz。这里使用的 NVDK 核心 DSP 为 C6416。体系结构如图 5.22 所示。

NVDK 具有强大的处理能力和丰富的外围设备，其主要特点包括 PAL 制和 NTSC 制视频的捕获和输出，CD 音质的音频记录与播放，强大的计算能力，能够实时处理图像压缩（JPEG，JPEG2000），视频压缩（MPEG-1，MPEG-4，H.263）或者音频压缩（MP3，AAC），也可以应用在需要很高计算能力的信号处理领域。

子板提供 I/O 接口、数字音视频接口、以太网口及模拟 I/O 端口等。NVDK 能够单独工作，也可以通过 PCI 总线接口作为 PC 的扩展卡工作。

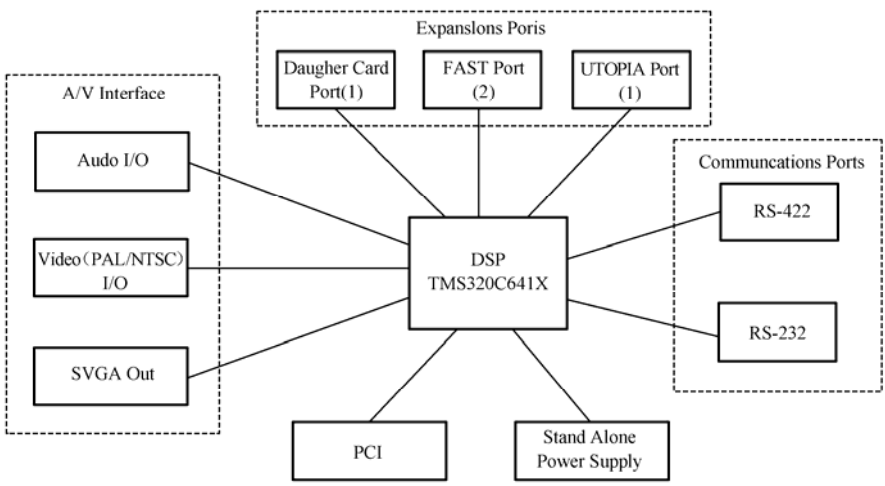


图 5.22　NVDK 板的体系结构

2. DSP 及其相关资源

TMS320C6415 及 TMS320C6416 均具有独立工作模式及 PCI 工作模式的功能，都具有 UTOPIA 端口。所不同的是，TMS320C6416 具有片上通信协处理器（VPC 及 TPC）。

1）扩展内存

DSP 上有两个扩展内存接口总线，分别为 EMIF-A 和 EMIF-B，EMIF-A 有 64 位宽度，EMIF-B 有 16 位宽度。分别运行在 100 MHz 频率下，特殊情况能够运行在 133 MHz 频率下。

外部内存依如下规定映射到接口总线上：

- SDRAM A 映射到 EMIF-A：256 MB，64 位宽度，最大瞬时速率能够达到 800 MB/s；
- SDRAM B 映射到 EMIF-B：8 MB，16 位宽度，最大瞬时速率能达到 200 MB/s；
- Flash 映射到 EMIF-B：4 MB，8 位宽度，按页访问（1 页 1 MB，共 4 页）。该 Flash 被分成 64 个 64 KB 的单元，由于按页访问机制，每次只能访问 16 个单元。

DSP 内存地址映射如表 5.2 所示。

表 5.2　DSP 内存地址映射表

名　　称	大　　小	CE	DSP 地址(Hex)
内部 RAM	1 MB		00000000..000FFFFF
内部寄存器			01800000..01C3FFFF
内部寄存器			02000000..02000033
McBSP 数据	256 MB		30000000..3FFFFFFF
SDRAM EMIF-B	4M×16(8 MB)	CEB0	60000000..601FFFFF
FLASH	1M×8×2 页(4 MB)	CEB1	64000000..641FFFFF
视频 FIFO	1×16(2B)	CEB2	68000000..68000001
快速端口 B FIFO	1×16(2B)	CEB3	6C000000..6C000001
SDRAM EMIF-A	32M×64(256MB)	CEA0	80000000..8FFFFFFF
支持逻辑	16×8(16B)	CEA1	90000000..9000007F
支持逻辑	16×8(16B)	CEA1	90010000..9001007F
支持逻辑	64×16(128B)	CEA1	90200000..902001FF
快速端口 AConfig bus	16×8(16B)	CEA1	90400000..9060007F
快速端口 B Config bus	16×8(16B)	CEA1	90600000..9060007F
互连平台接口 CEA	512K×32×2 页(4 MB)	CEA2	A0000000..A03FFFFF
互连平台接口 CEB	512K×32×2 页(4 MB)	CEA2	A0400000..A07FFFFF
快速端口 AFIFO	1×32(4B)	CEA3	B0000000..B0000007

2）复位和启动

NVDK 板上有两种复位：一是通过产生复位脉冲或采用手动按钮复位；二是通过 PCI 总线复位。前者对整个开发板，包括 DSP 内核（不包括 PCI 接口）进行复位，这意味着按下复位按钮，DSP 及其外设将会重启，但不会对 PCI 进行复位，开发板依然对 PCI 请求进行回应。PCI 复位方式由 PC 的 BIOS 控制，它将在 PC 启动或重启时重新配置 PCI 接口资源，PCI 复位不会影响 DSP 内核及其外设。

DSP 能够以两种模式启动：从 Flash 启动或者从 PCI 接口启动（只能应用于 PCI 模式）。第一种模式在独立运行模式下非常有用，它能够使应用很快启动。当然，此种模式也能应用于 PCI 运行模式下，使得 DSP 能够在 PC 完全启动以前工作。

3）Flash 的使用

当 DSP 运行在 Flash 启动模式下，重启 DSP 意味着激活其内部引导程序。内部引导程序将 Flash 内部最低位置的 1 KB 数据复制到 DSP 内存中地址的最低位置。这 1 KB 数据包括中断矢量表、PCI 模式下使用的通信表、二级引导程序（能够使用户将第一页剩余数据下载到 DSP 的内存中）。

3. 视频接口

NVDK 的视频模块能够捕获或播放符合 PAL 制及 NTSC 制的模拟视频，视频输入可以是复合视频格式或者 S-video 格式。视频捕获支持三种尺寸模式：全尺寸、CIF、QCIF。视频输出支持全尺寸，CIF 两种尺寸模式。各格式描述见表 5. 3。

表 5.3　视频格式

	PAL	NTSC
帧频	25 f/s	30 f/s
FULL	720 像素×576 行	720 像素×480 行
CIF	352 像素×288 行	352 像素×240 行
QCIF	176 像素×144 行	176 像素×120 行

图像数据的数字格式为 YUV422，即每个像素点用 16 位存储。在双字中，低端字节存储亮度信号，高端字节存储色度信号，奇数像素点存储色度信号中 U 的部分，偶数像素点存储色度信号中 V 的部分。

图 5.23 是视频模块的体系结构。

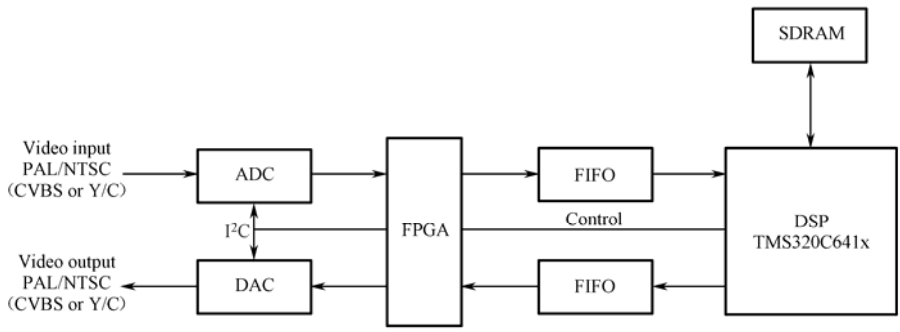


图 5.23　视频模块体系结构

1）视频捕获

从复合视频端或 S-Video 端输入的模拟视频，被输送到数字解码器（BT 835），将信号转

换成 YUV422 格式的数字视频，这些信号被写入视频捕获队列，DSP 将通过 DMA 读取这些信号。

FPGA 有两个目的：给队列写像素信息；其次是做时域抽取，将视频序列抽取成一个短视频，抽取比例可编程，从 1∶1 到 1∶63（每 63 帧抽取一帧）。

队列的作用是缓存数据流，使 DSP 能够连续读写数据。

2）视频产生

数字视频在内存中是 YUV422 格式的，DSP 通过 DMA 将数据写入输出队列，数据流将传送到输出编码器，在那里被转换成模拟视频信号（PAL 制或 NTSC 制），因为没有硬件帧存储器，即使是输出静止图像，视频帧数据必须被 DSP 不断刷新。

3）SVGA 输出格式

NVDK 的 SVGA 输出格式如表 5.4 所示。

表 5.4　SVGA 输出格式

源　图　像	实 际 图 像		输 出 图 像		垂直刷新速率	说　　明
	宽/pix	高/pix	宽/pix	高/pix		
CIF-NTSC	352	480	800	600	72 Hz（50 Mpix/s）	硬件将输出垂直加倍，达到 704/pix/行
CIF-PAL	352	576	800	600		
Full-NTSC	720	480	800	600		
Full-PAL	720	576	800	600		
SVGA	800	600	800	600		标准 SVGA
XVGA	1024	768	1024	768	70 Hz（75 Mpix/s）	标准 SVGA

4. 音频接口

NVDK 版上的音频模块具有立体声输入、输出及单声道麦克风输入。输入信号被音频编码器数字化为 16bPCM 格式，数字音频流通过 McBSP 串口进入 DSP。同样，输出信号也通过 McBSP 串口到音频编解码器，转换为模拟信号输出。音频编解码器由 FPGA 的控制逻辑通过 I^2C 串行总线进行配置。采样频率可编程，范围是 8～96 kS/s（样本点每秒）。图 5.24 为音频模块原理图。

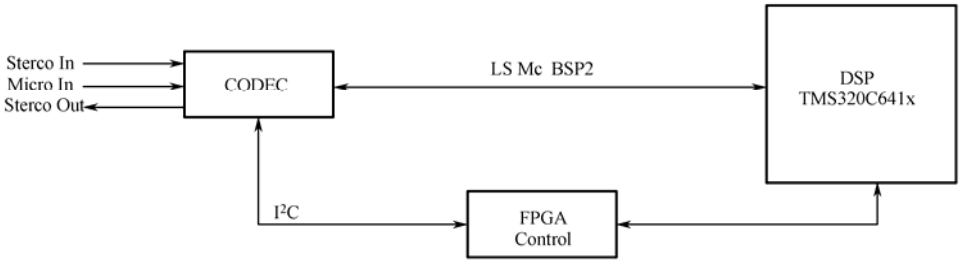


图 5.24　音频模块原理图

5. PCI 接口

NVDK 板具有 PCI 总线接口，该接口符合 PCI 标准 2.2 版。提供一条 32 位，33 MHz 的总线，信号电压适应 3.3 V 或 5 V 的总线标准。关于总线接口更多的信息，可以参阅 TI 的相关文档。

6. FAST-Port 接口

- 板上拥有两个 FAST-Ports（快速 ATEME 同步传输接口）接口。该接口用来使 NVDK 板能够同子板或其他系统快速交换数据。两个 FAST-Port 接口 A，B 通过同步双向队列分别与 DSP 的外围存储器 EMIF-A 和 EMIF-B 相连，图 5.25 是该接口的框图。

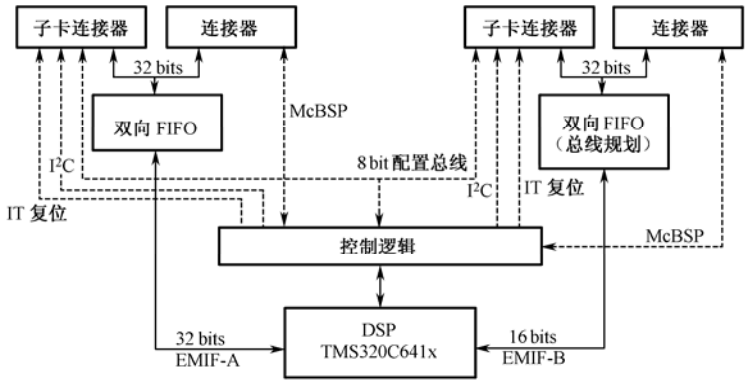


图 5.25　FAST-Port 接口框图

7. 其他接口

NVDK 有各种接口和其他板子及系统通信，包括子板，主要有：

- 两个 FAST（Fast ATEME Synchronous Transfer Port）接口，这种高速接口用于多板级联和多处理器的应用，能够扩展板子的输入、输出能力。
- 交叉平台子板接口，这种接口符合 TI 提出的交叉平台子板接口标准，许多板子都符合这种接口标准。
- UTOPIA 接口，用于同 ATM 连接的接口。
- RS-232 普通串口。
- RS-422，DSP 的 McBSP 串口符合 RS-422 标准，此类串口用于长距离传输。

5.6　H.264 在 NVDK 上的实现与优化

本节介绍将 H.264 Baseline Profile 的编码器在以 C6416 为核心的 DSP 开发平台 NVDK 实现上，通过优化，实现 QCIF（176×144）格式下的实时编码。

在 DSP 等嵌入式系统上的软件实现和 PC 上的软件实现有明显的区别。对于大部分基于 PC 的应用程序来说，只需要考虑如何编写代码就足够了。当然，在编写代码时要考虑算法

的性能和数据结构，其他像内存分配和 CPU 处理时间的分配就不用考虑。但是在嵌入式系统上实现应用程序，除了要考虑上述问题之外，还要考虑系统的初始化和存储空间的分配、CPU 的处理分配、并行化处理等问题。一般而言，在 DSP 上实现软件需要采用系统=程序+初始化+存储器管理的方式实现。

算法的实现与优化需要经过几个步骤：算法的选择、算法分析、代码移植及代码优化。下面逐一介绍项目的实现步骤，并着重介绍算法在 DSP 上的优化方法，最后给出实验结果。

5.6.1　算法选择

ITU 的官方网站上提供了很多可选的标准软件可以采用，包括 Tml 系列和 Jm 系列，因此选择一个合适的算法软件显得非常必要。本节采用 Jm6.1e 版本的 Baseline Profile 作为实现的基础，这么选择的理由一是相对于别的软件版本，Jm6.1e 是开始进行算法实现和优化工作时，ITU-T 提供的最新版本的参考模型，里面包含了 H.264 标准中最新的技术，具有最好的编码效果；二是由于 H.264 的算法复杂度过高，Baseline 中不包括一些复杂度很高的可选选项，而其采用的技术已经能够满足实现高效媒体处理平台的需要，因此，Baseline Profile Level 2.0 是实现的目标。

H.264 和以往视频编码标准类似，依然采用“运动估计补偿+变换编码+熵编码”的编码模式。其编码模块主要有运动估计、DCT 变换、熵编码、插值部分、环路滤波器等。它也遵循逐级编码，从帧级编码，到 Slice 级编码，最后到宏块级编码。

其编码器流程如图 5.26 所示，该图是使用 Intel VTune performance analyzer 软件分析出来的流程结构图。

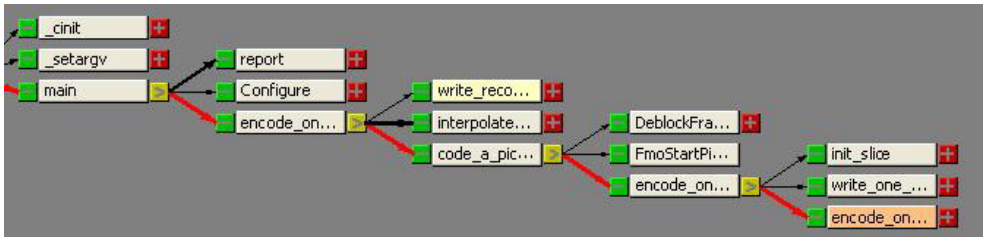


图 5.26　H.264 编码器流程

编码器主流程如图 5.27 所示。

在 Jm6.1 对协议的实现中，H.264 的 Main Profile 编码算法的具体实现可以由高到低分为四个层次。首先通过 main 函数入口进入编码程序，main 函数通过循环调用函数 encode_one_frame( )编码整个视频序列（如图 5.28 所示）；函数 encode_one_frame( )为编码算法的第二层，通过循环调用函数 encode_one_slice( )进行视频序列中每一帧的编码（如图 5.29 所示）；函数 encode_one_slice( )为编码算法的第三层，通过循环调用函数 encode_one_macroblock( )对每帧中的每一个 Slice 进行编码（如图 5.30 所示）；函数 encode_one_macroblock( )为编码算法的第四层，实现对每个宏块的编码（如图 5.31 所示）。

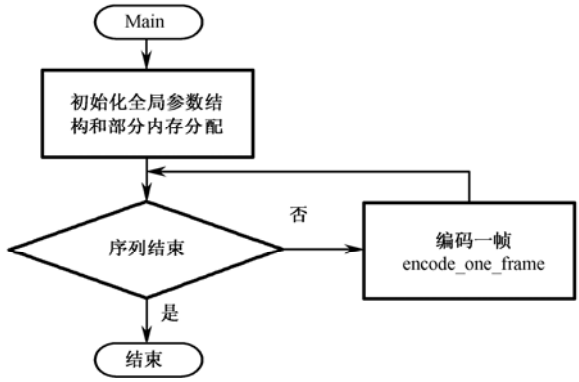


图 5.27　编码器主流程

主流程首先把系统的全局参数分配好，并为部分全局结构和参数分配内存空间，这里主要是 img、input 等全局参数中的信息，还有部分缓存器的内存分配。然后，循环调用 decode_one_frame 函数来编码一帧，并判断是否到达序列末尾，如果是则结束编码进程，否则循环编码，直到编码结束为止，这个循环是算法的主循环流程。

encode_one_frame 流程如图 5.28 所示。

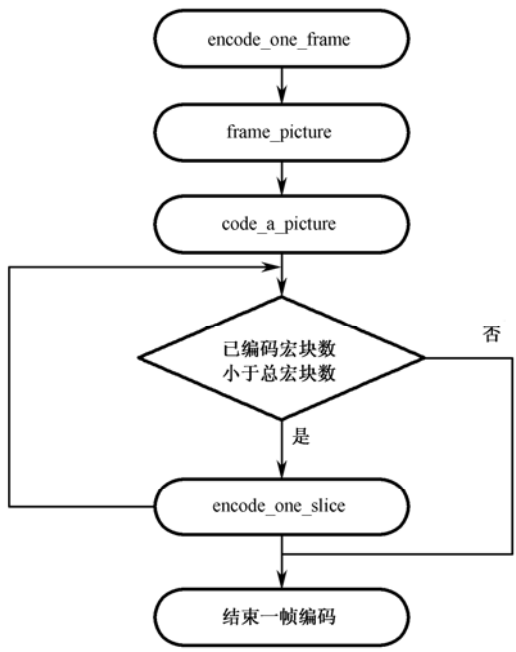


图 5.28　encode_one_frame 的流程图

encode_one_slice 流程如图 5.29 所示。

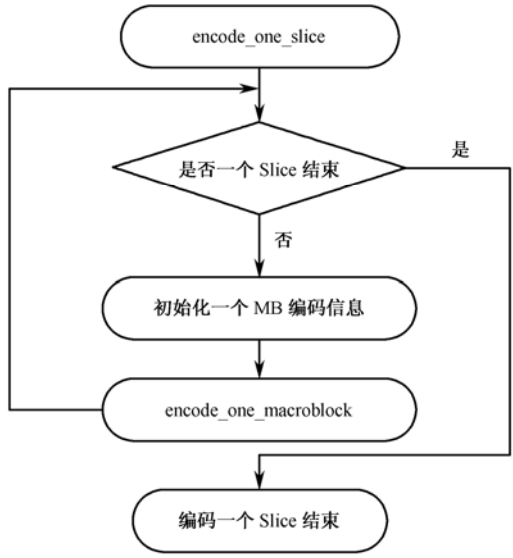


图 5.29　encode_one_slice 的流程图

编码一个宏块的流程如图 5.30 所示。

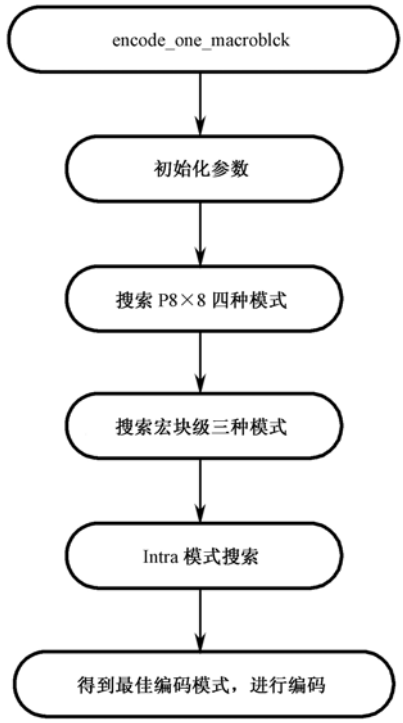


图 5.30　编码一个宏块的流程

5.6.2　编码器代码移植

代码的移植，顾名思义，就是将能在 PC 上运行的程序移植到 DSP 端，进行一定的调整，使其满足 DSP 的语法规范，能够初步运行。

代码的移植要做以下几部分工作。

1）冗余代码删除

ITU-T 公布的参考软件 Jm6.1e 是一个近 20 000 行，300 余个函数的大型程序，结构复杂，冗余度很高，里面有许多和 DSP 实现无关的代码，可以进行删除处理，以提高代码执行效率。例如，原始代码中有大量 debug 信息，trace 信息，assert 信息及 print 函数等，这些都是代码编写过程中调试所需要的信息，在实现到 DSP 端时，可以删除掉。此外原始代码中还有计算 SNR 的函数及大量的统计函数，运算量也颇大，而这些函数对以 Jm6.1e 为平台进行仿真研究时有意义，对于实现一个紧凑的编码系统则可以不予考虑，没必要由 DSP 来同步完成。

此外，由于 NVDK 板同 PC 通信通过仿真器由并口相连，数据传输速度较慢，编码后的码流传入 PC 存储也需要相当大的时间消耗，在具体实现中，通过在板上开辟一片缓存，存储压缩后的码流，待整个视频序列编码结束后，再统一将编码后码流传送到 PC 存储，这样和 PC 通信的时间就不占用编码的总时间。因为我们是在仿真平台上进行代码的实现与优化，完成 H.264 的实时编码算法内核，不涉及电路级的设计，等以后具体应用及系统设计的时候，DSP 和存储设备的通信可以通过 PCI 总线等其他方式进行大规模数据的快速传输。

2）内存空间分配

DSP 上的算法优化和 PC 端的一个重要不同就是 DSP 的片上内存资源非常有限，需要合理地进行分配。NVDK 板中，C6416DSP 的片上内存有 1MB，主要分四部分使用：系统中断矢量表、引导程序空间、PCI 通信表和用户使用区，其时钟频率可以达到和 CPU 同频，所以，如何合理利用内部存储区，是算法能否实时化的一个关键。片外内存分为 SDRAMA 和 SDRAMB，分别为 256 MB 和 8 MB。但是，由于片外存储器的工作时钟为 CPU 时钟的 4 分频或者 6 分频，所以，只能用来存储不经常读取的数据和一次性可以大量读取的数据（此时，可以打开 DMA，直接进行内存操作），同时，由于 SDRAMA 是 64 位的，这样大量的数据可以并行传送。

内存分配的原则就是常用数据及程序都分配到片上内存，以方便快速访问。不太常用的数据，以及数据量较大的数据分配到片外内存，例如，参考帧，以及捕获的视频序列等。表 5.5 是 NVDK 内存分布表。

表 5.5　NVDK 内存分布表

存储段名称	VECTORS	BOOTLOADER	COMMDEST	SDRAM B	SDRAM A
基地址	0x00000000	0x00000200	0x000003C0	0x60000000	0x80000000
长度	0x00000200	0x000001C0	0x00000040	0x00800000	0x10000000
用途	代码/数据	系统引导	系统	代码/数据	代码/数据
注释	复位和中断 矢量表	引导程序 预留空间	PCI 通信 描述子表格	外部 SDRAM EMIF B （8 MB）	外部 SDRAM EMIF A（256 MB）

3）代码的标准 C 化

TI 公司为 DSP 软件开发提供了名叫 Code Composer Studio（CCS）的功能强大的开发工具，作为应用软件软硬件调试仿真的集成开发环境。CCS 提供了基于标准 C 的编译器和优化器。通过 CCS 的 C 编译器优化器，标准 C 语言写成的代码就可以在 DSP 上进行编译运行，因此，需要将 C 语言代码标准 C 化，这涉及以下几方面的工作：

- 去除原始代码中的文件操作，因为对于在开发板上运行的 DSP 程序来说，没有文件的概念，只有存储空间的概念。原始视频流通过仿真器或摄像头捕获到板上缓冲区，编码完后的码流也存放在内存中。
- 规范数据类型，对于 C6416 来说，它是一款定点 DSP，支持的数据类型有限，不像 VC 中数据类型那么丰富。标准 C 化时需要对数据类型进行规范。
- 更改一些微软对标准 C 的扩展，例如，一些关键字的修改和去除。

标准 C 化后的代码，通过编译，能够在 DSP 上初步运行。

5.6.3 代码优化

通过代码移植能够获得在 DSP 上初步运行的代码，但是由于没有考虑到 DSP 自身的硬件特点，不适合 DSP 强大的并行处理能力，因此执行效率低下，不能满足实时要求，需要对其进行进一步优化。

对 DSP 代码进行优化的手段有以下三个层次，分别是项目级优化、算法级优化、指令级优化。下面对这三种优化手段分别进行介绍。

1. 项目级优化

项目级优化，是对项目的整体优化，主要手段有以下几点。

首先是在对整个项目进行编译链接生成 DSP 代码时，合理选择配置编译器选项，并针对这些参数选择，对程序进行调整和修正。其中进行的工作有：

- 项目编译时，通过参数-o3 调用最高级别的软件流水线优化，通过参数-mw 调用软件流水线循环反馈，从而增大软件编译成 DSP 代码的并行性。
- 项目编译时，用编译参数-pm，-o3 和-mt 来改善循环、多重循环、庞大循环体循环的性能。
- 只读变量声明成 const 型，循环计数器定义为 int 型，从而加大 DSP 代码的并行性。

其次对程序结构进行调整，对不适合 DSP 执行的语句进行改写，以提高代码的并行性。例如，DSP 并行性很高，能够对代码进行流水线处理，但是原始代码存在大量条件判断语句，会对流水线造成中断，不利于代码的并行处理，因此，可以采用判断提前，去除不必要的判断等方式减少判断语句对流水线的中断。

2. 算法级优化

算法级优化，就是利用 H.264 的自身特点，采用一些快速算法，在不影响编码质量的前提下，提高编码器速度，从而在速度和质量上达到一个较好的平衡。

这部分内容会在 5.7 节专门介绍，这里就不赘述了。

3. 指令级优化

上述方法无法达到实时要求，就要进行指令级优化。C64x 系列 DSP 有丰富的具有高度并行处理能力的指令。下面介绍一些 C64x 系列 DSP 媒体处理的相关指令。

- ADD4　加法指令，一次执行 4 对 8 位数据的加法。一个寄存器有 32 位，可以存放 4 个 8 位数据。计算中，两个源寄存器中的 4 组对应 8 位数据分别相加，结果存放在目标寄存器中。
- AVGU4　一次执行 4 对 8 位无符号数据求平均运算。计算中，2 个源寄存器中的 4 组 8 位无符号型紧缩字求平均，结果以 4 个 8 位紧缩字的形式存放在目标寄存器中。
- DOTPU4　一次执行 4 对 8 位无符号数据点乘运算。计算中，2 个源寄存器中的 4 组 8 位无符号型紧缩字对应相乘，乘积相加，所得结果存放在 32 位寄存器中。
- SUBABS4　一次执行 4 对 8 位无符号数据求差绝对值运算。计算中，2 个源寄存器中的 4 组 8 位无符号型紧缩字对应相减，差值求绝对值，所得结果以 4 个 8 位紧缩字的形式存放在目标寄存器中。
- LDB/LDH/LDW/LDDW　将 8 位、16 位、32 位或 64 位数据读入目标寄存器中，所读取的数据在内存中是地址 align（32 位对齐）的数据。
- LDNW/LDNDW　将一个 32 位或 64 位的非对齐数据读入目标寄存器中。
- STB/STH/STW/STDW　将 8 位、16 位、32 位或 64 位数据写入内存中，所写入的数据在内存中是地址 align（32 位对齐）的数据。
- STNW/STNDW　将一个 32 位或 64 位的非对齐数据写入内存中。

以上是 DSP 丰富的媒体处理指令的一部分。由于视频处理都是以像素为单位进行计算，而一个像素值的大小是 8 位数据。视频处理中需要频繁对像素进行加、减、乘、求绝对值等运算，这些并行指令能够极大地加速计算速度，再加上 DSP 具有 8 个独立功能单元，理论上一个指令周期，同时最多能够处理 8 条指令，这样的媒体处理能力是相当可观的。

指令级优化是利用 DSP 的特点和相关并行指令，对代码进行优化的手段。主要方法有：

1）循环拆解，排流水

C6416 DSP 是并行化程度很高的一款 DSP 芯片，但是，如果程序中判断跳转过多，或者程序的循环嵌套深度过大，将严重影响到 DSP 发挥其并行效果。另外，DSP 的并行效果还反映在其对循环的软件流水分配上，但是，DSP 软件流水的限制很多，其中一条就是只对最内层循环做软件流水分配，这样，循环嵌套太多，就不利于流水线的分工。所以需要对程序内部所有的两层或者三层循环嵌套进行拆解，拆解循环过程并不是随意的，而是要充分利用 DSP 芯片的特性。下面以 DCT 变换中的一段函数为例，讲解如何进行循环拆解。

```
for (j=0; j < BLOCK_SIZE; j++) //其中，BLOCK_SIZE=4;
{
  for (i=0; i < 2; i++)
  {
    i1=3-i;
    m5[i]=img->m7[i][j]+img->m7[i1][j];
    m5[i1]=img->m7[i][j]-img->m7[i1][j];
  }
```

```
    img->m7[0][j]=(m5[0]+m5[1]);
    img->m7[2][j]=(m5[0]-m5[1]);
    img->m7[1][j]=m5[3]*2+m5[2];
    img->m7[3][j]=m5[3]-m5[2]*2;
}
```

这是 DCT 变换函数中水平变换的一段代码。在外层大循环中，还嵌套着内层循环，影响软件流水分配，需要将内存循环打开。此外，代码中还使用了数组定义临时变量，在 DSP 编译过程中，无论寄存器是否有剩余，集合形式的变量都会被分配到内存中，影响数据的读写速度。通过对代码的观察，发现临时变量较少，可以将数组型变量用独立变量替换。将 m5[]换成 m50～m53，img->m7[][]为全局变量，不处理，修改后的代码如下：

```
for (j=0; j < BLOCK_SIZE; j++) //其中，BLOCK_SIZE=4;
{
   m50=img->m7[0][j]+img->m7[3][j];
   m53=img->m7[0][j]-img->m7[3][j];
   m51=img->m7[1][j]+img->m7[2][j];
   m52=img->m7[1][j]-img->m7[2][j];
   img->m7[0][j]=(m50+m51);
  img->m7[2][j]=(m50-m51);
  img->m7[1][j]=m53*2+m52;
  img->m7[3][j]=m53-m52*2;
}
```

程序中类似的多重循环还很多，通过对一些耗时较大部分的多重循环进行拆解，能够较大提高代码的执行效率。

2）使用内联函数

TI 的 CCS 开发环境提供了丰富的内联函数，内联函数就是对线性汇编指令的函数封装。内联函数可以方便地在 C 代码内部进行调用，不影响代码的整体结构，还能提高代码的执行效率。下面以一段读取内存的函数为例，讲解内联函数的使用。

```
for (j=0; j < BLOCK_SIZE; j++)
  for (i=0; i < BLOCK_SIZE; i++)
   imgY[img->pix_y+block_y+j][img->pix_x+block_x+i]=img->m7[j][i];
```

这段代码的作用是将一个 4×4 块的数据，从变量 img->m7 复制到 imgY，类似这样的读写内存运算在程序中频繁出现，需要逐个字节进行读取、存储。

转化为汇编语言为：

```
  LDB img->m7[i][j],A
  STB A, imgY[img->pix_y+block_y+j][img->pix_x+block_x+i]
```

读这些内存指令需要 4 个时钟周期来完成，粗略估算复制 16 B 的数据所需要的时钟数为（4+1）×16=80 个时钟周期。

内联函数_mem4(void * ptr)允许一次完成对 4 B 数据的读写操作，这样上面的代码可以改写为：

```
          for (j=0; j < BLOCK_SIZE; j++)
        {
            data=_mem4(&(img->m7[j][0]));
_mem4(&imgY[img->pix_y+block_y+j][img->pix_x+block_x+0])=data;
        }
```

这样一来，这个循环只需要（4+1）×4=20 个时钟周期，只是原来时钟数的 25%。

在代码中，插值、运动估计、DCT 变换、反 DCT 变换等部分都涉及大量数据的复制，使用内联函数对提高代码执行效率效果明显。

3）调整数据结构

从前面分析可以知道，视频处理中，频繁的内存访问是运算量消耗大的一个重要因素，因此，如何提高内存访问速度，成为代码优化、提高编码器效率的一个不可回避的问题。通过调整数据结构，将需要大规模访问的数据在内存地址中连续存放，这样方便使用并行性指令对内存访问，提高访问内存的速度。

下面以插值函数为例，介绍如何进行数据结构的调整。H.264 算法中采用 1/4 精度的插值，对于 QCIF 序列，一帧 176×144 大小的图像，插值后以（176×4）×（144×4）大小的矩阵存放在内存中，如图 5.31 所示。在亚像素运动估计中，如果当前块与参考块进行亚像素位置的 SAD 计算，对应点在内存中的存储位置将不连续（横向或纵向均间隔 4 个亚像素位置点），这样，读取亚像素位置的像素值将只能够逐个点进行，无法用并行性指令进行访问，这样的访问效率非常低下，严重影响内存访问的速度。通过更改数据结构，将亚像素点在内存中以 16×（176×144）的矩阵进行存放，即将亚像素点分为 16 类，同一性质的亚像素点归为一类，存放在一个 25 344 B 长度的连续内存中，这样，在亚像素搜索中，对应位置点在内存中物理地址连续，可以利用并行性指令，一次读取 4 个或 8 个点的像素值。

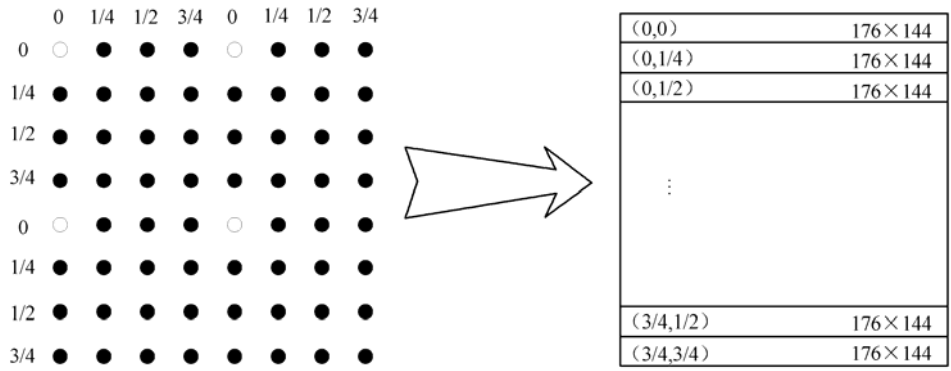


图 5.31　插值函数数据结构调整示意图

此外，还可以用 NVDK 自带的 EDMA 函数 DAT_open，DAT_copy 等对大块数据进行复制，如将重建图像输出到片外参考帧缓存中。使用 DMA 函数能够独立于 CPU 操作，将 CPU 从繁重的内存复制工作中解放出来。

4）线性汇编改写程序

还可以将一些耗时较大或调用次数较多的函数抽取出来，使用线性汇编语言进行改写，运用媒体处理指令，手动排流水，充分利用 DSP 指令及运算单元的并行性，尽量一个时钟周期完成更多的指令，从而提高代码的执行速度。涉及的函数模块有：DCT 变换、反 DCT 变换，整像素运动估计，亚像素搜索，帧内编码函数，插值函数等，优化效果明显。

下面以运动估计模块中最常用的 SAD 计算为例，讲解如何使用线性汇编语言进行优化。

前文已经提到，运动估计部分是视频编码器中耗时最大的模块，而运动估计使用的标准是 SAD 最小原则，涉及大量的 SAD 计算，如式 5.1 所示。

$$\mathrm{SAD}=\sum_{i=0}^{N-1}\sum_{j=0}^{M-1}|C_{ij}-R_{ij}| \tag{5.1}$$

其中，M，N 是块的维数，$C_{i,j}$ 是被估计块第 i 行第 j 列的像素值，$R_{i,j}$ 是参考块第 i 行第 j 列的像素值，可见一次 SAD 运算中需要大量的加法、减法和绝对值运算。不同大小块的 SAD 计算量是不同的，对于 8×8 尺寸的块来说，需要进行 64 对点的读取、求差、绝对值、求和运算。

C64xDSP 的 LDDW 和 LDNDW 一次可以将 8 B 的 align 或 non-align 数据读入寄存器，这样，8×8 块的一行数据一条指令就能读入寄存器中。

此外，SUBABS4 指令能够同时计算 4 B 数据的差绝对值运算。DOTPU4 指令能够计算 4B 数据的点积运算，上一指令所得到的绝对值结果，通过点积 0x01010101 掩码，就能够得到绝对值之和，因此，SAD 值的计算过程可以如图 5.32 所示。

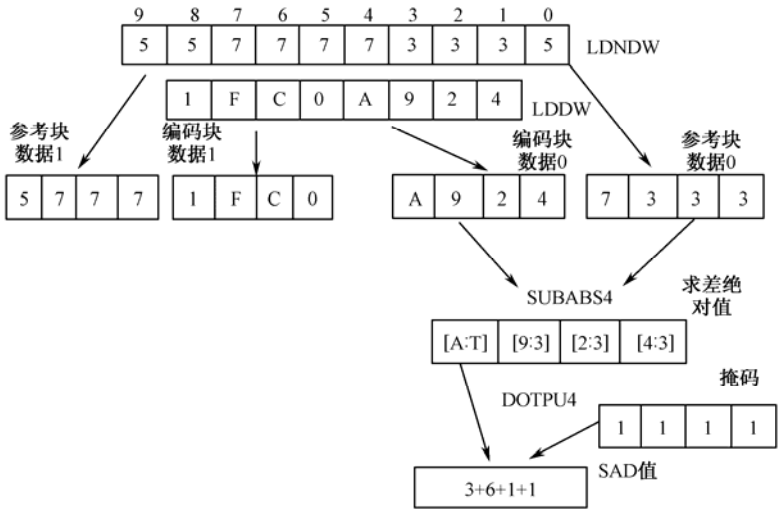


图 5.32　SAD 值的计算过程示意图

优化后的 8×8 块一行数据的 SAD 计算只需要 7 条指令（2 条 load 指令，2 条 SUBABS4 指令，2 条 DOTPU4 指令，1 条求和指令）就能够完成，大大节约了运算量。

5.6.4　程序优化结果

通过以上的优化手段，编码器的效果如下：编码器对 QCIF（176×144）序列，中等运动下，每秒钟编码 40～50 帧。对于 CIF（352×288）序列，中等运动下，每秒钟编码 10 帧。

编码器实验采用 3 个 H.264 标准测试序列，包括 CIF 和 QCIF 序列，共 20 帧，采用 IPPPP…编码模式。表 5.6 为编码后 PSNR、编码帧率、码率等实验结果，可以看出，优化后的编码器具有较好的性能。

表 5.6　编码器优化实验结果

视频序列（20 帧）	平均峰值信噪比 PSNR/dB			总编码时间/ms	帧率/（f/s）	码率/（kb/s）
	亮度 *Y*	色度 *U*	色度 *V*			
Bus（CIF）	34.87	40.19	40.86	2137	10	1 457.3
Container（QCIF）	36.35	41.22	40.91	426	47	72.2
Foreman（QCIF）	35.67	39.12	40.64	476	42	136.9

5.7　算 法 优 化

H.264 仍然使用“运动估计+变换编码+熵编码”的模式，不同之处是采用了很多特殊的技术以降低码率，提高性能，例如，1/4 精度的亚像素运动矢量搜索，多帧预测，7 种模式的块匹配，以及帧内预测，整数 DCT 变换，CABAC 或 CAVLC 的熵编码技术，等等。这些技术的使用，使得 H.264 具有低码率、高质量的性能，可以说是视频编码领域的一个优秀的标准。但是这些技术的使用也大大增加了编码器的复杂度，使得运算量大大增加。如何对 H.264 的算法进行优化，使其满足实际工程需要是一个研究热点。

运动估计是视频编码算法中的关键模块，它的性能优劣对编码后的码流大小及图像质量都有着至关重要的影响。图 5.33 显示了 H.264 编码器（Jm6.1e）采用快速全搜索算法时各个模块运算量在总运算量中所占的比例。可以看出，整像素和亚像素运动估计部分在整个编码器中所占运算量比例很大，此外，内存读写部分的运算量很大部分也是在运动估计运算时消耗的，因此，运动估计模块在编码器计算量中所占比例超过 70%，提出高效、快速的运动估计算法，成为对 H.264 编码器优化中的关键问题。

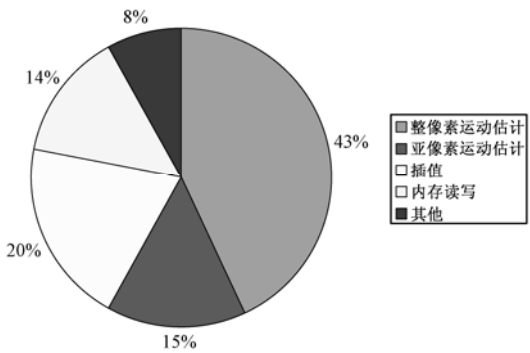


图 5.33　H.264 编码器各模块运算量示意图

H.264 编码器运动估计模块中所包含的具体技术有：整像素运动估计，亚像素运动估计及帧间编码的 7 种块大小选择技术等。在本节中，将根据 H.264 算法的自身特点，针对以上问题，研究新的快速整像素运动估计算法，快速亚像素运动估计算法，以及快速宏块模式选择算法，显著地降低运动估计模块的运算消耗，提高编码器的运行速度。

5.7.1　快速整像素运动估计算法——ARPS-4

近年来出现的一些针对 H.264 优化的运动估计算法，利用周边块信息进行运动矢量搜索成为一种重要的思想。2003 年提出的 ARPS-3（Unequal-arm Adaptive Rood Pattern Search）是其中性能较好的一个算法，平均速度比 DS（Diamond Search）算法提高两倍多。此外，早停止技术在近两年也成为运动估计算法研究中的热点，通过设定固定的或自适应的阈值，使得当匹配误差达到要求的范围以内，就可以结束搜索。早停止技术在 MV-FAST（Motion Vector Field Adaptive Search Algorithm），以及 PMV-FAST （Predictive Motion Vector Field Adaptive Search Technique）等算法中运用，有良好的效果。

本节将基于自适应阈值的早停止技术运用于 ARPS-3 算法中，提出一种新的运动估计算法 APRS-4，其性能比 APRS-3 有显著提高，远远高于 DS、新三步法等经典快速搜索算法，图像质量并没有明显影响。

1）早停止技术

早停止技术的基本思想可归结为：寻找足够好的匹配，而非寻找最佳的匹配。通过设定一个阈值，在当前搜索点的匹配误差低于这个阈值时，停止搜索，认为这个匹配点就是所需要的匹配点。通过这种方法，减少搜索点的个数，达到加速搜索的目的。

显然，阈值如何选取，是一个非常重要的问题，阈值太高，则搜索精度太差，影响图像质量；阈值太低，无法达到加速的目的。目前，一些算法都是通过经验选取一个固定阈值，如 MVFAST 算法，但是固定阈值的缺陷也显而易见，无法对不同视频序列进行自适应调整。为了解决这个问题，可利用相邻块之间残差场的空域相关性及相同位置块在相邻帧之间残差场的时域相关性，根据上一帧及已编码块的信息进行阈值设定，从而达到自适应的目的。阈值的选取可以描述为：

$$T_k = f_k(\mathrm{MSAD}_1，\ \mathrm{MSAD}_2，\ \cdots，\ \mathrm{MSAD}_i，\ \cdots，\ \mathrm{MSAD}_n) + b$$

其中，T_k 是当前所求阈值，MSAD_i 为相关参考块的最佳 SAD 值。f_k 则为相关块的最佳 SAD 值的函数表达式，一般采用线形组合或最小值、中值等。参考块一般选取周边已编码的邻块及上一帧的当前位置块。b 为常数，当其较大时，搜索速度快，但搜索精度就要差一些；当其较小时，搜索精度较好，但是搜索速度就要慢一些。算法可以根据不同实际需要对 b 值进行设置。

2）APRS-3 算法简述

为了方便下面的论述，先将 APRS-3 算法简要进行描述。

第 1 步【预测】：基于相邻块的运动矢量进行中值预测，当块大小为 8×16 或者 16×8 时，采用方向分割预测模式，其他块大小使用中值预测模式，如图 5.34 所示，预测结果作为当前搜索块的预测运动矢量。一般预测矢量为：

pred_mv = median(Mv_A, Mv_B, Mv_C)

详细方法可见 H.264 文档的中值预测部分。

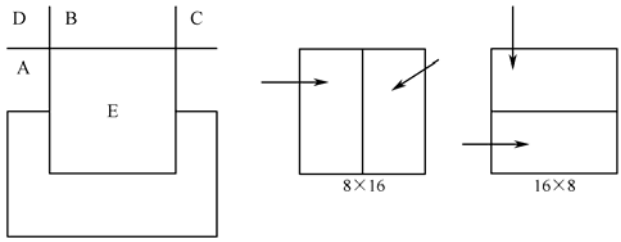


图 5.34　运动矢量预测方法

第 2 步【初始化搜索点】：非对称十字形搜索图样的中心点位于第 1 步的预测运动矢量处，即 $\mathbf{MV}_0$ 位置。十字图样的 4 个顶点的矢量选取规则如下：

$$\begin{aligned}\mathbf{MV}_1&=\{\max(\mathrm{MV}_x),\mathrm{MV}_{\text{predicted-y}}\}\\ \mathbf{MV}_2&=\{\min(\mathrm{MV}_x),\mathrm{MV}_{\text{predicted-y}}\}\\ \mathbf{MV}_3&=\{\mathrm{MV}_{\text{predicted-x}},\min(\mathrm{MV}_y)\}\\ \mathbf{MV}_4&=\{\mathrm{MV}_{\text{predicted-x}},\max(\mathrm{MV}_y)\}\end{aligned}\qquad(5.2)$$

其中，MV_x,MV_y 是周围邻块运动矢量的水平和垂直分量。然后对搜索图样中的 5 个点及当前块的原点（$\mathbf{MV}_5=(0,0)$）进行搜索，如图 5.35 所示，这一步只进行一次。

第 3 步【精细搜索】：以上一步的最佳匹配点为搜索中心，进行小菱形图样的搜索，如图 5.36 所示，直到最佳匹配点在菱形的中心，则该点为最佳匹配点，得到的矢量为该块的运动矢量。此外，如果第 2 步中十字形图样的臂长均小于或等于 1，则直接进行第 3 步。

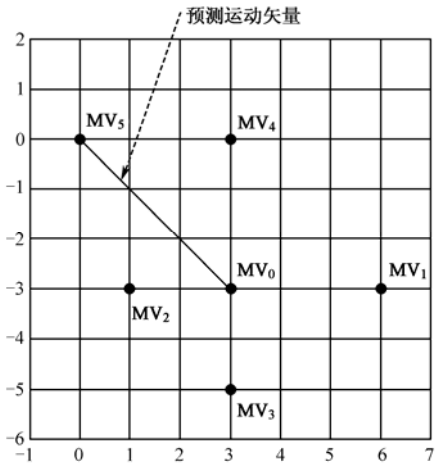


图 5.35　非对称十字形搜索图样

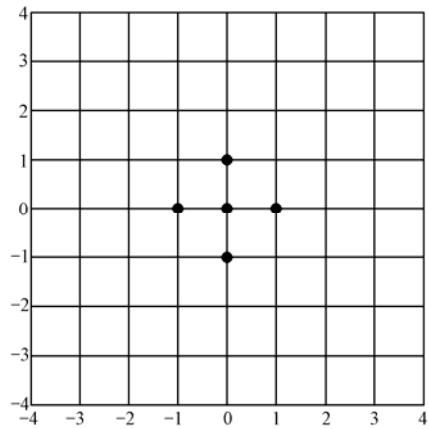


图 5.36　精细搜索小菱形图样

3）改进的非对称十字形图样搜索算法 ARPS-4 描述

新算法在 APRS-3 算法基础上做了两点改进：一是设置了自适应的搜索误差阈值。二是设置一个队列，记录已经搜索过的点，避免重复搜索。

算法描述如下所述。

第 1 步【预测】：方法如同 APRS-3 算法。

第 2 步【初始化搜索点】：搜索图样如图 5.35 所示，搜索矢量的计算如式（5.2）。当搜索矢量为 $\mathbf{MV}_0$ 时，设置搜索误差阈值 T_1，T_1=2Np+b，其中 Np 为当前搜索块的像素点个数，b 为常数，这里设置为 20。当搜索误差 SAD 小于阈值 T_1 时，则停止搜索，该矢量即为所要求的运动矢量。如果搜索误差高于阈值，则继续搜索周围其他 5 个点，此时设置搜索误差阈值 T_2，T_2 计算如下：

$$T_2 = \min(\mathrm{MSAD}_1,\ \mathrm{MSAD}_2,\ \cdots,\ \mathrm{MSAD}_i,\ \cdots,\ \mathrm{MSAD}_n, 3Np) + b \qquad (5.3)$$

其中，MSAD_i 为周围已编码相同大小邻块的搜索残差值。为了保证搜索精度，将周边邻块搜索残差值中的最小值和三倍当前块的像素个数进行比较，其中的最小值和 b 的和作为阈值 T_2。当搜索误差小于 T_2，则停止搜索，当前搜索矢量满足要求。否则，进行下一步搜索。

第 3 步【精细搜索】：搜索图样为小菱形，如图 5.36 所示，当搜索误差小于阈值 T_2，则停止搜索，当前结果满足要求，否则，继续搜索，直到菱形中心为误差最小点。此外，如果第 2 步中十字形图样的臂长均小于或等于 1，则直接进行第 3 步。

5.7.2　基于早停止技术的亚像素运动估计快速算法

实验结果显示，采用了 ARPS-4 技术后，整像素运动估计每个块平均搜索点数已经缩小到 3～5 个点，而 H.264 采用逐级精确的亚像素运动估计算法，在 1/4 精度搜索时需要 16 个点，1/8 精度搜索时需要 24 个点，和整像素相比，运算开销已经很大。因此，提出一种高效快速的亚像素运动估计算法十分必要。

1）亚像素全搜索算法

H.264 标准中采用的是全搜索的亚像素运动估计算法。与整像素不同，亚像素运动估计是逐级精确的，类似整像素亚采样与快速分层算法，在得到最佳匹配整数像素点后，搜索 1/2 像素位置，得到最佳点后，搜索 1/4 像素位置，直到得到所需要的精度。

如图 5.37 所示，圆形点代表整数像素，三角形点代表 1/2 像素，方形点代表 1/4 像素，其中（0, 0）点为最佳整像素位置。搜索时，先搜索最佳整像素位置周围的 8 个 1/2 像素点，找到最佳 1/2 像素位置，然后再搜索最佳 1/2 像素点周围的 8 个 1/4 像素点，找到最佳匹配点，从而得到亚像素的运动矢量。

如果进行 1/2 像素精度的搜索，全搜索需要搜索 8 个点，1/4 精度全搜索，需要搜索 16 个点，依次类推，进行 $1/2^n$ 精度搜索时，需要搜索 8n 个点。

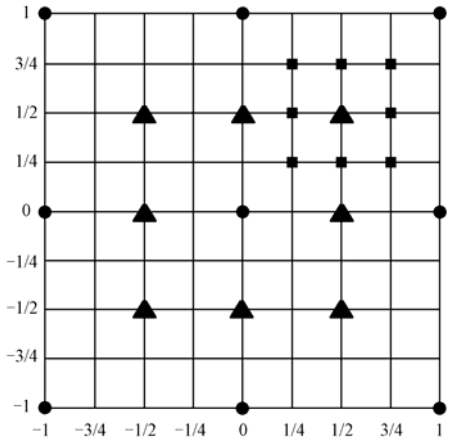


图 5.37　亚像素全搜索示意图

2）亚像素快速搜索算法介绍

亚像素快速搜索算法主要基于两点考虑，以提高搜索速度。

一是利用亚像素点空域相关性，缩小搜索范围，减少搜索点数。因为亚像素是通过对整像素的插值产生，是对整像素的一种平滑。在 H.264 中，1/2 像素插值是通过一个 6 抽头滤波器实现的，1/4 像素插值是对周围像素的简单平均，因此，在亚像素级别，像素点之间具有较强相关性，这就是本算法设计的一个出发点。通过试验发现，水平和垂直方向四个点对最后码率的影响较大（见表 5.7）。因此，在亚像素每一级搜索中，本算法首先搜索垂直水平方向四个点，从中找到最佳匹配点，然后搜索与该点相邻的两个斜向亚像素点，以达到提高精度的目的。如在第一步搜索中，1 点（见图 5.38）为最佳匹配点，则进一步搜索其相邻的 7，5 两个点，从中找到最佳匹配点，以进行下一级搜索。

二是引入了早停止技术，亚像素搜索采用比整像素略低的阈值，并设置两级阈值。在 1/2 精度搜索时，当误差小于 $0.9T_2$ 时（T_2 见式（5.3）），则停止整个搜索，当误差小于 $0.95T_2$ 时，停止当前 1/2 精度搜索，进入 1/4 精度搜索。进行 1/4 精度搜索时，当误差小于 $0.9T_2$ 时，则停止整个搜索。

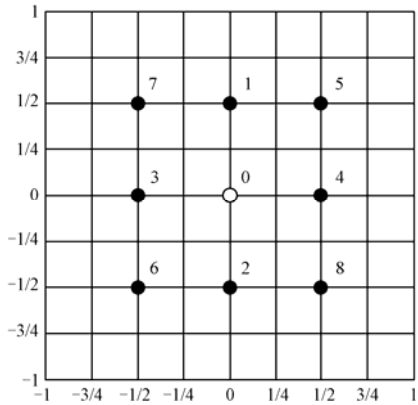


图 5.38　亚像素快速搜索算法示意图

表 5.7　亚像素搜索对码率的影响（kb/s）*

测 试 序 列	亚像素全搜索	去掉亚像素搜索	只搜索垂直水平位置四个点	只搜索其他位置四个点
Football	1 359.06	2 397.48	1 391.68	1 465.94
Foreman	116.33	403.52	122.34	147.19
Container	45.45	79.15	45.57	78.52

*说明：本实验在 H.264/AVC 的参考模型 Jm6.1e 上实现，编码 100 帧，帧率为 30 Hz，搜索区域为±16，单帧参考，1/4 精度搜索，使用全部 7 种块模式，量化参数为 28，去掉 R-D 优化选项。编码模式为 IPPP…，整数像素采用全搜索。

3）亚像素快速搜索算法步骤描述

第 1 步：首先搜索最佳整数匹配点周围 1，2，3，4（见图 5.39）四个位置的 1/2 精度亚像素点。当匹配误差低于 $0.9T_2$ 时，该点即为最佳亚像素点，结束全部搜索。当匹配误差低于 $0.95T_2$ 时，则停止搜索，该点即为 1/2 精度匹配点，进入第 3 步，否则继续搜索。

第 2 步：搜索第 1 步中获得的最佳匹配点相邻的两个点，得到的最佳匹配点作为 1/2 精度最佳匹配点。当匹配误差低于 $0.9T_2$，该点即为最佳亚像素点，结束全部搜索。当匹配误差低于 $0.95T_2$ 时，则停止搜索，该点即为 1/2 精度匹配点，进入第 3 步，否则继续搜索。

第 3 步：搜索 1/2 精度最佳匹配点周围 1，2，3，4 四个位置的 1/4 精度亚像素点。当匹配误差低于 $0.9T_2$ 时，该点即为 1/4 精度匹配点，整个搜索过程结束。否则继续搜索。

第 4 步：搜索第 3 步中获得的最佳匹配点相邻的两个点，得到的最佳匹配点作为 1/4 精度最佳匹配点。当匹配误差低于 $0.9T_2$ 时，则停止搜索，该点即为最佳亚像素匹配点，否则继续搜索。

5.7.3　快速运动估计算法实验结果与分析

我们在目前 H.264/AVC 的参考模型 Jm6.1e 上实现本节提出的 ARPS-4 算法及亚像素快速搜索算法。测试环境为 Intel Celeron 2.4G CPU。选择有代表性的 8 个标准测试序列：Football，Flowergarden（352×240），Tempete（352×288），Foreman，Salesman，Container，Grandma，News（176×144），其中 Football，Tempete 为大运动序列，Foreman，Salesman 为中等运动序列，Container，Grandma，News 为小运动序列或几乎静止序列，Flowergarden 为具有较多细节和镜头平移序列。测试时，帧率为 30 Hz，搜索区域为±16，单帧参考，1/4 精度搜索，使用全部 7 种块模式，量化参数为 28，去掉 Rd 优化选项。编码模式为 IPPP…，编码 100 帧。为了便于比较，除了实现全搜索算法（FS）外，还实现了 ARPS-3，新三步法（NTSS），菱形算法（DS）。

对实验结果进行统计发现，快速算法和 FS 相比，亮度分量的 PSNR 损失非常小。本节给出的快速算法具有良好的适应性，无论在大运动、小运动、还是镜头移动或缩放的序列中，最大 PSNR 损失不超过 0.1 dB，平均损失为 0.035 dB，这样的损失对于运动估计快速算法来说可以是忽略不计的，说明该快速算法对图像质量有较好的保持。

对统计快速算法和 FS 在速度方面的性能进行比较，本节提出的快速算法，搜索速度比 ARPS-3 算法有了明显提高，远远高于 NTSS 和 DS 算法，加速比平均达到 270 倍左右。对于大运动序列和具有较多细节及镜头平移的序列，在 PSNR 损失很小的情况下，仍然具有较高的

搜索速度。此外，亚像素快速搜索算法平均搜索速度比全搜索提高 1 倍，节省运算量近 50%，加速效果十分明显。

通过实验给出快速算法和 FS 相比的码率大小，从统计数据可以看出，ARPS-4 算法性能好于 NTSS 算法、DS 算法，略低于 ARPS-3 算法，平均仅增加码率 1.71%。

本节提出的两种快速搜索算法，充分利用了块之间运动矢量的相关性，并结合基于自适应阈值的早停止技术，具有较好的性能。整像素搜索平均每块搜索点数为 4 个多点（加速比 270 余倍），亚像素搜索快速算法和 1/4 精度全搜索相比，节省运算量 40%～60%。此外，提出的算法和全搜索相比，保持了较好的图像质量和较低的码率。在大运动、中等运动、小运动、多细节和镜头移动缩放等情况下，新算法均保持了良好的性能。由于算法平均每块搜索点数仅为几个点，为避免重复搜索需要记录的点也仅为几个，因此，这部分额外内存和计算开销是很少的。

5.7.4　快速模式选择算法

以往的视频编码标准如 H.263，MPEG-1，MPEG -2，MPEG -4 等只采用一种或两种块大小模式，而 H.264 采用从 16×16 到 4×4 不等的 7 种块大小模式。这种技术使得 H.264 相对于 H.263 有大约 20%的码率节省，但是产生的运算量消耗也是很可观的。为了实现实时编码，快速模式选择算法成为关键性的问题。

近一两年来，有一些学者围绕这个问题展开研究，发表了一些论文。大致思路有两个：一种思路是利用宏块的空域相关性，利用周边宏块信息对当前宏块的模式进行预测，但是由于模式的选择对最终码率影响比较大，预测的不精确会带来码率的较大增加，因此这种方法效果并不太好；另一种思路基于不同像素精度的运动估计结果来对模式进行选择，能够显著减少亚像素精度的搜索点数，但是对整像素搜索点数没有降低，因此还有进一步提高的余地。

本节吸收了以上方法的优点，并结合一些新的技术，提出一种快速模式选择算法。这种算法主要基于以下方法：模式预测技术，基于自适应阈值的早停止技术，后搜索技术及基于整像素或半像素精度的模式早确定技术。实验结果显示，本节的方法对模式选择模块的运算开销有较大降低，并能够较好地保持图像的编码质量和码率。下面将对这种算法进行详细介绍。

1. H.264 中的模式选择算法

H.264 中存在 7 种不同大小的块模式，它们被分成两类：宏块级块模式，包括 16×16，8×16，16×8 三种模式；亚宏块级块模式，也称为 P8×8+模式，包括 8×8，8×4，4×8，4×4 等块大小模式。

在模式选择中，被选中的最优模式必须有最小的费用函数值，费用函数公式如下：

$$J(\boldsymbol{m},\lambda_{\text{motion}}) = \text{SAD}(s,c(\boldsymbol{m})) + \lambda_{\text{motion}} \cdot R(\boldsymbol{m} - \boldsymbol{p})$$

其中，$\boldsymbol{m} = (m_x, m_y)^{\text{T}}$ 是当前的运动矢量；$\boldsymbol{p} = (p_x, p_y)^{\text{T}}$ 是预测运动矢量；$R(\boldsymbol{m} - \boldsymbol{p})$ 代表用于编码运动矢量信息的比特数；SAD$(s,c(\boldsymbol{m}))$ 代表当前宏块和参考宏块的 SAD 值；λ_{motion} 是拉格朗日乘法算子。

从上面的公式可以看出，费用函数值由两部分信息构成，残差场信息和运动矢量等边信息。如果选择较大尺寸的块大小模式，则用于对运动矢量信息编码的比特数减少，但是残差

场将有较高的能量，将耗费较多的比特数进行编码。特别是在一些具有较多细节，较大运动的宏块内，选择较小尺寸的块大小模式，将使得残差场能量降低，这部分码率消耗将降低，但是将需要耗费更多的比特数对运动矢量及宏块模式信息进行编码，这部分码率开销将会增大。因此，如何选择最佳的块模式对编码器的性能有着很大的影响。

在 H.264 的参考软件 Jm6.1e 中，通过计算所有模式的费用函数选择最优模式，步骤如下所述。

第 1 步：首先计算宏块中 4 个 8×8 大小子块的费用函数。计算每个 8×8 子块的所有 P8×8+模式（8×8，8×4，4×8，4×4）的费用函数，其中最优的模式为 8×8 子块的模式，费用函数作为该子块的费用函数，所有 4 个 8×8 子块的费用函数之和，作为该宏块在 P8×8+模式下的费用函数。

第 2 步：计算 16×16，8×16，16×8 三种模式的费用函数，和 P8×8+模式的费用函数作比较，值最小的模式作为当前宏块的最终编码模式。

2. 快速模式选择算法

本算法基于以下 4 点技术：

1）模式预测技术

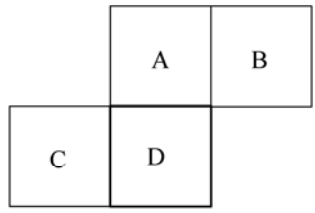
因为相邻宏块间存在很高的相关性，使用周边邻块的信息对当前块的信息进行预测是可行的，这一思路已被应用到运动矢量的预测上，成为 H.264 标准的一部分。同样，也可以将这种思路应用于模式选择中，通过对当前宏块编码模式进行预测，以提高编码速度。

表 5.8 显示了宏块与周边邻块间的模式匹配度，这里选取了三个测试序列，分别是大运动序列 Flower，中等运动序列 Foreman，小运动序列 Grandma，参考软件 Jm6.1e，采用 IPPP…编码模式，整像素运动估计采用全搜索。可以看出，其相关性还是很明显的。

表 5.8　宏块与周边邻块间模式匹配度

	Flower	Foreman	Grandma
匹配度	81.77%	89.59%	93.38%

图 5.39 是周边邻块分布示意图，被用来预测的邻块包括上邻块，左邻块，右上邻块，以及上一帧的当前位置块。



A: 上邻块
B: 右上邻块
C: 左邻块
D: 上一帧当前位置宏块

图 5.39　周边邻块分布示意图

从上面的讨论可以看出，当前宏块和周边邻块有较大的相关性，用周边邻块的模式对当前宏块的模式进行预测是可行的，公式如下：

$$\text{CurrentMode}=\text{Min\_cost}(\text{mode}A,\text{mode}B,\text{mode}C,\text{mode}D)$$

这样，每一宏块至多搜索 4 种模式，和原先搜索 7 种模式相比，运算量有了下降。

2）早停止技术

同样，将基于自适应阈值的早停止技术应用于快速模式选择算法中，设置一个自适应阈值 T_1，在模式选择过程中，某一模式的费用函数值低于该阈值，则认为该模式已经足够好，满足要求，可以作为编码模式，停止对其他候选模式进行搜索，从而节省搜索时间。自适应阈值的选取为：

$$T_1=\min(\min\text{cost}\_A,\min\text{cost}\_B,\min\text{cost}\_C,\min\text{cost}\_D) \tag{5.4}$$

这里，阈值 T_1 等于各参考宏块最小费用函数值中的最小值。

表 5.9 是各种级别块大小被选为最优模式的百分比。在这里，同样选取 Flower，Foreman，Grandma 三个测试序列，编码方式和前文一致。从中可以发现，较大尺寸的宏块被选中的概率较大，这也和绝大多数视频序列静止或准静止区域在画面中所占比重较大相一致。因此，为了使自适应阈值更加有效，我们在模式搜索时可以交换宏块级模式和 P8×8 级模式的搜索顺序，先搜索较大尺寸的模式，使得自适应阈值命中概率更大一些。

表 5.9　各级别模式成为最优模式的百分比

	MB level	P8×8	Intra
Flower	63.74%	34.81%	1.45%
Foreman	91.93%	6.87%	1.20%
Grandma	98.35%	0.65%	1.00%

3）基于整像素与半像素的模式早确定技术

在原始模式选择算法中，为了找到最优模式，需要计算 1/4 像素精度下的费用函数值，这样的方法虽然精确，但是却增加了运算复杂度。为了解决问题，要进行调整，方法描述如下：

对于 P8×8 模式，仅仅需要计算在整像素精度下的费用函数，具有最小费用函数值的模式作为最佳的 P8×8 级别的模式，然后再计算该模式下 1/2 和 1/4 精度运动估计，得到该模式精确的费用函数值。

对于宏块级模式，因为大尺寸模式对最终编码结果影响较大，需要在 1/2 像素精度下计算费用函数，比较找到最优宏块级模式，然后进一步计算该模式下 1/4 精度的费用函数。

将宏块级最优模式和 P8×8 级最优模式的费用函数值比较，较小的模式将是最终的最优模式。

该算法一定程度上降低了亚像素运动估计搜索点数，从而提高了模式选择的运算速度。

4）后搜索技术

因为模式的选取对于最终编码结果有着较大的影响，因此，模式预测显得尤为重要。在某些场景中，预测的候选模式并不能正确反映当前编码宏块的信息，这样将会使得最后码率

上升，编码质量下降。例如，周边邻块均为静止或准静止的区域，将使用大尺寸模式进行编码，而当前编码宏块具有较大运动或较多细节，适宜用较小尺寸模式进行编码，这时，就不能正确预测当前宏块的编码模式。从表 5.7 中也可以看出，当前块和参考块的模式也存在一定的失配比率，这在大运动视频序列中尤为突出。

为了解决这个问题，我们采用一种后搜索技术，设置一个自适应阈值 T_2，当所有预测模式的费用函数值均高于这个阈值时，我们认为预测模式不能正确反映当前编码块的情况，需要对其他所有未被搜索的模式进行搜索，从而找到最佳模式，自适应阈值的计算是：

$$T_2 = \max(\min\cos t\_A, \min\cos t\_B, \min\cos t\_C, \min\cos t\_D) \quad (5.5)$$

基于以上的技术，这里给出一种新的快速模式选择算法，算法描述如下所述。

第 1 步：模式预测。

当编码宏块位于图像第一行或第一列，所有 7 种模式均需要被搜索。

当编码宏块位于图像的右边界，只需要用 A，C，D（如图 5.40 所示）的模式进行预测。

如果编码宏块位于图像中间，则 A，B，C，D 宏块的模式将作为编码宏块的候选模式。

第 2 步：搜索。

首先搜索候选模式中宏块级模式，然后再搜索 P8×8 模式。设置一个阈值 T_1，如式（5.4）所示，如果当前模式的费用函数值低于阈值，则停止搜索，该模式作为该宏块的最佳编码模式。否则，搜索所有候选模式，比较它们的费用函数值，确定最佳模式。如果宏块位于图像左边和上边界，阈值将被设置为 0，如果宏块位于右边界，只使用宏块 A，C，D 来计算阈值。

对于 P8×8 模式，最佳模式在整像素精度下获得，对于宏块级模式，最佳模式在半像素精度下获得。

第 3 步：后搜索。

设置一个阈值 T_2（如式（5.5）所示）。如果步骤 2 中的费用函数值均高于该阈值，则需要搜索候选模式外的所有其他模式。

3. 实验结果

在 H.264 参考软件 Jm6.1e 中实现本节的新算法，并和全模式搜索算法相比较，选择三个不同的测试序列，分别为大运动序列 Flower（SCIF 352×240），中等运动序列 Foreman（QCIF 176×144）和小运动序列 Grandma（QCIF 176×144）。每种序列均编码 100 帧，运动估计算法采用前面提出的 ARPS-4 算法。对于编码选项，关闭 R-D 优化选项，不使用 B 帧，单帧参考，搜索区域±16，帧率 30Hz，编码模式为 IPPPP…。

1）速度

表 5.10 给出了三种测试序列，我们提出的快速模式选择算法和原始模式选择算法的各种模式搜索点数的比较及运算量节省的比率。从表中可以看出，我们提出的快速算法对搜索点数的节省，尤其是亚像素搜索，效果是明显的。

表 5.10　快速模式选择算法加速效果

（a）　Flower（SCIF）

	快速模式选择算法		非快速算法		节省搜索点数百分比	
模式	整像素	亚像素	整像素	亚像素	整像素	亚像素
16×16	217 882	416 504	254 765	522 720	14.5%	20.3%
16×8	227 702	303 792	524 065	1 045 440	56.6%	70.9%
8×16	171 372	217 552	534 236	1 045 440	67.9%	79.2%
8×8	634 198	988 704	1 040 550	2 090 880	39.1%	52.7%
8×4	1 205 830	268 480	2 003 257	4 181 760	39.8%	93.6%
4×8	1 216 180	212 704	2 016 206	4 181 760	39.7%	94.9%
4×4	2 190 771	272 192	3 790 019	8 363 520	42.2%	96.7%

（b）　Foreman（QCIF）

	快速模式选择算法		非快速算法		节省搜索点数百分比	
模式	整像素	亚像素	整像素	亚像素	整像素	亚像素
16×16	64 048	148 680	64 966	156 816	1.4%	5.2%
16×8	36 449	50 528	140 306	313 632	74.0%	83.9%
8×16	37 448	52 992	139 626	313 632	73.2%	83.1%
8×8	80 468	166 304	279 600	627 264	71.2%	73.5%
8×4	153 978	24 992	548 292	1 254 528	71.9%	98.0%
4×8	152 929	28 224	544 986	1 254 528	71.9%	97.8%
4×4	254 604	9 408	982 553	2 509 056	74.1%	99.6%

（c）　Grandma（QCIF）

	快速模式选择算法		非快速算法		节省搜索点数百分比	
模式	整像素	亚像素	整像素	亚像素	整像素	亚像素
16×16	48 367	156 680	48 495	156 816	0.3%	0.0%
16×8	17 819	32 688	101 411	313 632	82.4%	89.6%
8×16	17 638	32 400	101 268	313 632	82.6%	89.7%
8×8	33 548	125 200	195 945	627 264	82.9%	80.0%
8×4	46 401	4 064	346 345	1 254 528	86.6%	99.7%
4×8	45 864	7 424	343 355	1 254 528	86.6%	99.4%
4×4	68 047	256	595 622	2 509 056	88.6%	99.9%

下面进一步分析快速模式选择算法的加速性能。序列 Flower 包含较大的运动和较多的细节信息，因此使用 P8×8 模式更多一些，以使编码质量更好，因此快速算法对大尺寸模式的运算节省较大。对于那些中等运动或小运动序列来说，如 Foreman 和 Grandma，因为包含较多静止或准静止块，采用大尺寸的模式较多一些，于是快速算法对小尺寸模式计算量节省很

明显。这些都从实验结果中反映出来的。可见，这里提出的快速模式选择算法能够显著减少某些模式中不必要的运算，较大降低计算冗余。

由于 SAD 的计算是运动估计中的主要部分，因此在不同模式下一个搜索点对应的运算量是不同的。一般情况下，可以认为 $M\times N$ 模式下（M、N 等于 16、8 或者 4），一个搜索点的运算量是 16×16 模式下一个点运算量的 $M\times N/256$。以此为基础，可以将搜索点数的节省转换为运算量的节省，见表 5.11 所示。可以看出，快速算法对于亚像素与整像素运动估计运算量的节省是显著的。

表 5.11　运算量节省百分比

运算量节省百分比	Flower	Foreman	Grandma
亚像素	72.6%	77.3%	79.8%
整像素	43.0%	62.9%	71.9%
合计	62.9%	72.7%	78.0%

2）性能

我们看到快速模式选择算法可以极大地提高编码器的运行速度，同时也保持了编码器的编码性能几乎不受任何影响。表 5.12 给出了三个测试序列的编码结果对比。

表 5.12（a）　Flower 编码结果

	快速模式选择算法				非快速模式选择算法			
QP	24	28	32	36	24	28	32	36
SNR/dB	36.61	33.15	29.48	25.99	36.62	33.15	29.48	26.01
码率/（kb/s）	3 461.1	2 229.1	1 314.6	690.7	3 443.1	2 217.0	1 295.1	688.5

表 5.12（b）　Foreman 编码结果

	快速模式选择算法				非快速模式选择算法			
QP	24	28	32	36	24	28	32	36
SNR/dB	37.98	35.20	32.38	29.88	38.00	35.21	32.41	29.88
码率/（kb/s）	232.1	121.7	65.8	39.6	229.2	119.6	65.0	39.2

表 5.12（c）　Grandma 编码结果

	快速模式选择算法				非快速模式选择算法			
QP	24	28	32	36	24	28	32	36
SNR/dB	39.15	36.36	33.71	31.42	39.15	36.37	33.73	31.43
码率/（kb/s）	74.1	38.3	20.4	11.2	73.4	37.8	20.0	11.0

图 5.40 是根据表 5.12 绘制的三种编码的 R-D 曲线，从图中可以看到，快速模式选择算法较好地保持了图像的质量。

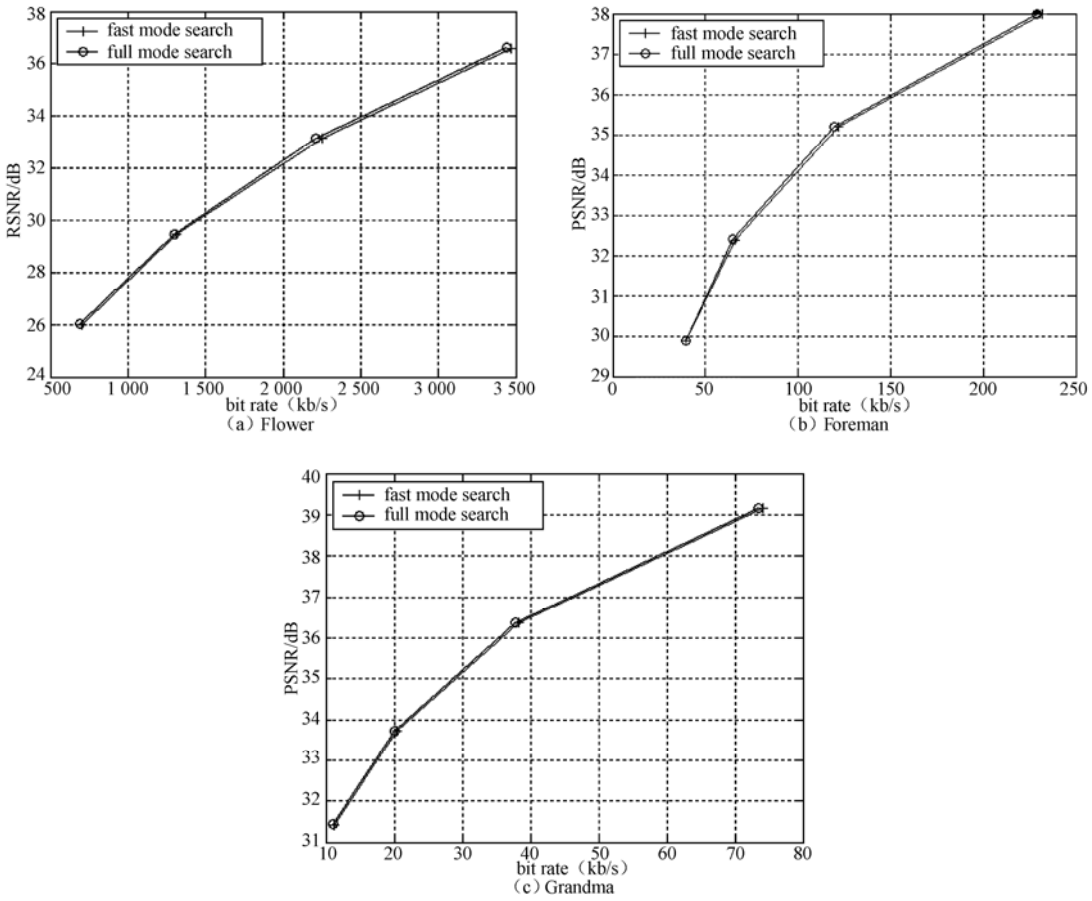


图 5.40　三种编码的 R-D 曲线

本节介绍了 H.264 在运动估计算法上的特点，并根据这些特点提出了一些快速算法，这些算法大大提高了编码速度。其中快速整像素运动估计算法每块平均搜索点数为 4～5 个，和全搜索相比，当搜索窗大小为 16 时，加速比为 250 倍左右，若搜索窗大小为 32，加速比可达 1 000 倍左右。快速亚像素搜索算法能够节约运算量 40%～60%。快速模式选择算法能够节约运算量 60%～80%。实验结果显示，这些算法在加快编码速度的同时，对编码质量（PSNR）没有大的影响，码率也没有显著增加，能够较好地保持 H.264 的编码性能。将这些算法改进，替代 DSP 实现中的相应模块，可以进一步提高编码器的编码速度。

本章介绍了在 TMS320C6416 平台上实现一个 H264 编码器的优化过程，包括算法的选择和算法优化、向 DSP 的移植、在 DSP 上的优化，这是一个很典型的开发过程。尽管，目前人们可能更多地关注 H264 在 DM642 或 Davinci 上的实现，由于 Davinci 和 DM642 均是 C6416 作为其 DSP 核，故本章详细的介绍对于希望进入这个领域的技术人员仍有参考价值。

参 考 文 献

［1］魏振宇．H.264 快速算法研究及其在高速 DSP 平台上的实现［D］．北京：清华大学电子工程系，2005.

［2］UB Video Inc.TMS320C64x Digital Media Platform Implementation, www.ubvideo.com.

［3］Iain E.G.Richardson．H.264 and MPEG-4 Video Compression, John Wiley & Sons Inc., 2003.

［4］S.Wenger. H.264/AVC over IP, IEEE Trans. Circuits Syst. Video Technol., vol.13, pp.645-656, Jul.2003.

［5］T.Stockhammer, M.M.Hannuksela, and T.Wiegand. H.264/AVC in wireless environments. IEEE Trans. Circuits Syst. Video Technol., vol.13, 657-673，Jul. 2003.

［6］钟玉涿，王琪，贺玉文. 基于对象的多媒体数据压缩编码国际标准——MPEG-4. 北京：科学出版社. 2000.

［7］张旭东，卢国栋，冯健. 图像编码基础和小波压缩技术——原理、算法和标准. 北京：清华大学出版社. 2004.

［8］S.Zhu and K.-K.Ma. A new diamond search algorithm for fast block-matching motion estimation. IEEE Trans. On Image Processing, vol.9, no.2, 287-290, Feb.2000.

［9］A.M. Tourapis, O.C. Au and M.L. Liou. Predictive Motion Vector Field Adaptive Search Technique (PMVFAST)-Enhancing Block Based Motion Estimation, Proc. Visual Communications and Image Processing 2001(VCIP-2001), 883-892, San Jose, CA, Jan. 2001.

第 6 章　H.264 编码器在 TMS320DM642 上的实现和优化

在这一章中，我们将叙述如何将 H.264 编码器的 Baseline Profile 移植到 TMS320DM642 EVM 上，并对此进行优化。最终实现超过 CIF 格式下的实时编码并达到 D1 格式下的准实时。有关 H.264 的主要原理请参考 5.1～5.4 节，此处不再重复。

6.1　TMS320DM642 EVM 介绍

TMS320DM642 EVM 是一个能供用户为 TI C64xx 系列 DSP 评估和开发应用程序的低功耗开发平台，其系统框图如图 6.1 所示。该 EVM 采用了专用媒体处理芯片 DM642，其核心是一个 C64x 内核。DM642 与 C6000 系列 DSP 的代码全兼容，方便代码移植。

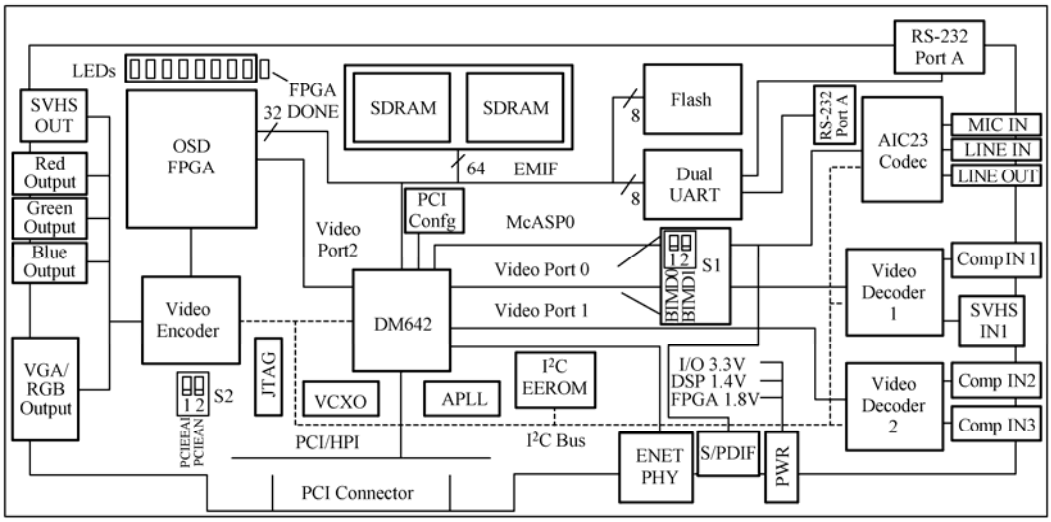


图 6.1　TMS320DM642 EVM 的系统框图

中央处理模块是整个硬件系统的核心，既负责运算处理，又担负着对周边其他模块的支撑和控制作用。此模块包括 TMS320DM642、256MB SDRAM 芯片组成的片外存储器及一片 Flash 存储器。

从 TMS320DM642 的视频输入口输入的视频图像首先被缓存到外部的图像数据存储器中，在处理的时候才被读入内部存储器。考虑到视频输入、输出过程需要多帧图像缓冲区进行暂存，同时视频处理算法需要复杂而庞大的数据结构来支撑其实现，因此系统配置了 256 MB

的 SDRAM 存储器。相对于 SRAM，SDRAM 具有存储容量大，价格便宜的优点，适合图像数据的存取。DM642 的 EMIF 接口为外部扩展的 SDRAM 存储器提供了非常好的支持，只需要简单配置若干个时序寄存器即可实现与 SDRAM 的接口。对于符合 PC-133 标准的 SDRAM，DM642 的 EMIF 可以支持高达 100 MHz 的数据存取速率。

Flash 存储器则保存了系统的 Boot Loader 程序及一些用户数据，当系统上电之后，DM642 从 Flash 调入程序进入片内存储器中运行，在对整个系统硬件进行自检和初始化配置后，DSP/BIOS 实时内核开始工作，按优先级从高到低对系统的各种处理任务进行调度。

TMS320DM642 是专门为视频应用而设计的数字信号处理器，其运算内核建立在 C64x 基础上，除了拥有众多支持视频和图像处理的特殊指令之外，还在片上集成了一些专门针对视频应用的外围设备和接口，从而免去了以前需要在视频编解码器与 DSP 之间增加 FIFO 缓冲器和粘连逻辑的麻烦，大大增加了系统的集成度，降低了功耗、体积，缩短了产品的上市时间。

DM642 核本身基于 TI 公司推出的高性能、先进的 Veloci TI 超长指令字（VLIW）结构。一个时钟周期可以并行执行最多 8 条 32 位指令，它将指令合并成一个最大 256 位宽的指令包，需要注意的是这里执行指令包长度并不总是 256 位的，它是变长的，可以根据当前时钟周期同时执行的指令个数而变化，这样有效地节约了缓存，这也是 C64 的 CPU 区别于其他微处理器超长指令字结构的显著特征。此外，C64x 将 Veloci TI 结构扩展成 Veloci TI.2 结构，该扩展使得在一个指令周期之内能够进行更多工作，从而在诸如视频和图像领域的应用中提高工作效率。

DM642 的内核结构可参见 5.5.2 节，这里不再赘述。

6.1.1　DM642 的缓存结构

DM642 在数字处理领域具有很高水平的性能，一方面，DM642 CPU 结构的紧密耦合及出色的编译器使之有最大的处理能力，类似 RISC 结构的指令集和流水线应用的扩展，使得许多指令具有高度的并行性；另一方面，高性能的两级缓存结构设计，使得数据吞吐量这个瓶颈得以缓解，CPU 能够运行在最高的速率。两级缓存通过片上片外数据的自动交换，降低了程序的运行时间，高性能的 EDMA 控制器也使 CPU 具有高效率。

图 6.2 所示为 DM642 的缓存结构，DM642 的 CPU 直接与一级程序缓存（L1P）和数据缓存（L1D）相接。这两个缓存大小各为 16 KB，能被 CPU 以最高频率访问。此外，一个二级程序/数据缓存（L2）可以为数据存储提供更多的灵活性。在 DM64x 系列中，L2 缓存的大小可以随时变化。片上 SRAM 共有 256 KB，为数据和程序共享空间，可以通过类似以下的代码进行配置，将其一部分空间作为二级 Cache 使用（最大可以划分 256 KB 的空间）。

示例 6-1　L2 缓存配置。

```
CACHE_setL2Mode(CACHE_256KCACHE);
```

在实际应用中，L2 缓存映射的 SRAM 一般用于存放视频流数据和代码的重要部分，以保证最高的传输速率。

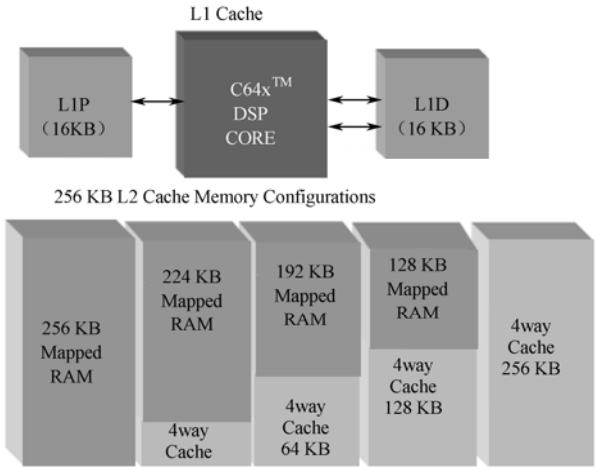


图 6.2　DM642 的缓存结构

6.1.2　TMS320DM642 的视频接口

区别于 TI 过去其他系列的 DSP，TMS320DM642 的最大特点就是片上的外围设备中集成了三个相互独立的视频接口（Video Port），每一个视频接口都可以自由配置为视频采集模式、视频显示模式或传输流接口采集模式。视频接口结构框图如图 6.3 所示。

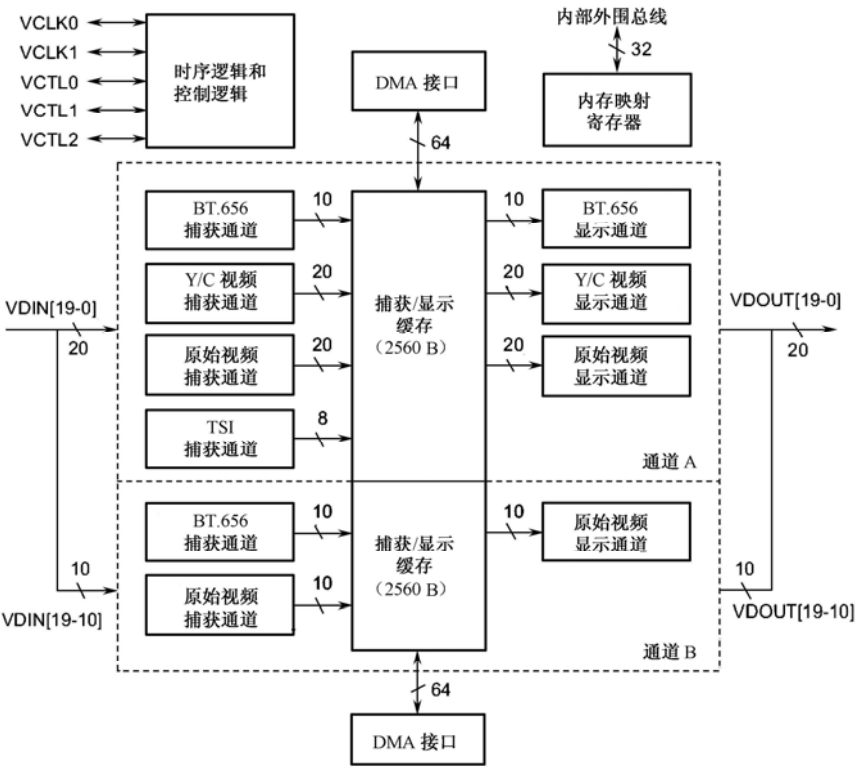


图 6.3　TMS320DM642 的视频接口结构框图

从图 6.3 视频接口的结构框图可以看出，一个视频接口被划分成两个独立的通道 A 和通道 B，每个通道各有 10 位的数据宽度，它们可以合并在一起成为一个 20 位的数据通道，或者在某些情况下单独使用。除了数据通道之外，视频接口还提供了外部时钟信号输入、输出（VCLK0 与 VCLK1）和控制信号输入、输出（VCTL0、VCTL1 与 VCTL2）接口，用于视频数据信号的同步。DM642 的视频接口可以跟业界的大部分视频编解码芯片进行接口，实现标准视频的输入、输出，另外通过裸数据通道，还可以实现与非标准视频设备的接口。

视频接口本身配备了 5 120 B 的 FIFO 缓冲区，根据模式的不同，这些缓冲区可以被进一步划分为多个小的缓冲区，每一个缓冲区又都跟 DMA 通道直接连接，当缓冲区的数据量高于（对于采集模式）或者低于（对于显示模式）某一个阈值的时候，则自动触发 DMA，完成在 FIFO 缓冲区与一个预先指定的帧缓存之间进行数据交换。

以视频采集模式为例，随着视频数据的输入，FIFO 缓冲区中的数据量达到某个阈值，从而触发 DMA 通道传输缓冲区的数据到用户指定的位置，比如在 SDRAM 中分配的一块内存区域。最后，当 DMA 将一帧完整图像的数据都传输到该内存区域的时候，DMA 触发中断，通知用户已经准备好一帧图像数据，用户得到该中断之后，则可以在中断响应程序中对该帧图像数据进行进一步的处理。

当视频接口配置为视频采集模式时，其输入时钟的频率可以达到 80 MHz，输入的视频数据流可以是 YUV422，Y/C 或者裸数据格式，同步的方式可以选择是内嵌的 SAV/EAV 同步或者是利用行、场同步信号进行外部同步；当视频接口配置为视频显示模式时，其输出数据的速率能够达到 110 MHz，输出数据的格式同样可以是 YUV422，Y/C 或者是裸数据格式，还可以根据视频编码芯片的需要，输出内嵌的同步信号或者是输出外部的行同步、场同步信号或者帧同步信号。

除此之外，视频接口还可以配置为传输流接口（TSI）采集模式，用于捕获传输码流。

6.2　DSP 平台的程序开发问题

在 DSP 等嵌入式系统上的软件实现和 PC 上的软件实现有着明显的区别。对于大部分基于 PC 的应用程序来说，只需要考虑算法的性能和数据结构，其他的如内存分配和 CPU 处理时间的分配基本上不用考虑，一般来说都由操作系统（如 Linux 或者 Windows）提供了良好的接口，但是在嵌入式系统上实现应用程序，这些问题都无法回避，同时还要考虑并行化处理等更低层的问题，这不仅要求对算法原理有深刻的理解，更要求能够掌握目标平台的硬件特性。通常 DSP 平台算法开发的难点有以下几点。

1）DSP 内部资源限制

DSP 的内部资源是十分有限的，例如，DM642 的片内存储器容量只有 256 KB 大小，不能将程序代码和待处理数据全部放到片内存储器中。为了提高代码的运行效率，只能将经常访问的程序代码和数据放到有限的片内 RAM 中进行处理，而其他数据则在必需的时候才调入内存中。这就要求必须要有可靠的数据传输机制，保证内外部存储器之间数据的高速搬移。

此外，由于 DM642 是定点数字信号处理器，内部没有配备浮点运算单元，因此在编译

时，程序中所有的浮点类型数据的运算都将被编译器转换为定点运算，这意味着运算量成倍增加，因此在算法设计时必须在计算精度和运行速度上寻找平衡。

2）指令并行性的问题

一般来说 DSP 的处理器频率并不高（相对于通用处理器），如 DM642 的最高频率为 720 MHz，其最大优势在于并行化，如在 DM642 的 CPU 内核中包括了 8 个相互独立的功能单元，这 8 个功能单元又被分为两组，分别与 32 个寄存器连接。理论上这 8 个功能单元在单个时钟周期内最多可以并行执行 8 条指令。但在实际中是很难达到这样的峰值运算速度的，程序设计人员只能试图接近这一目标。因此，在设计算法、编排程序代码的时候，应充分分析各功能单元的利用率，最大限度让指令并行执行。

3）算法流程的改进

在 PC 平台上编写代码的时候，通常不需要考虑数据在底层器件之间的传输过程，但是算法移植到 DSP 过程中，由于内核访问外部 SDRAM 的速度比直接访问片内 SRAM 的速度慢数倍，因此，需要改变算法的实现流程，改变原有的数据传输和数据处理串行进行的执行结构，实现数据传输和数据处理的并行处理，减少内核消耗在数据读取上的时间。

6.3　编码器实现

在 DSP 开发过程中，一般先在 PC 端进行算法设计仿真，随后再在 DSP 端实现优化，这一节主要讲述如何将已有的 PC 端编码器程序往 DSP 上移植的过程及要注意的问题。

6.3.1　算法基本流程

我们选择实验室已有的 PC 端调试成功的 H.264 编码器，其主流程如图 6.4 所示。

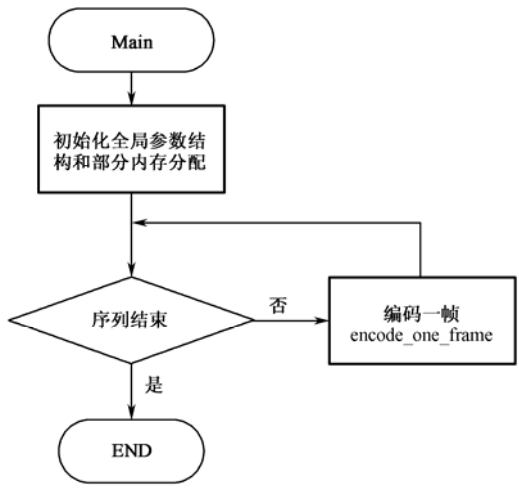


图 6.4　H.264 编码器主流程

从图 6.4 可以看到，H.264 首先通过 main 函数入口进入编码程序，main 函数通过循环调用函数 encode one frame() 编码整个视频序列。于是，我们的程序设计是，首先把系统的全局参数分配好，这里主要是 img，input 等全局参数中的信息，如在 main()函数中调用 tskProcessStart 函数进行初始化。

```
ThrProcess thrProcess;
……
void tskProcessStart() {
……
CHAN_open( &thrProcess.chanList[ chanNum ], &thrProcess.cellList[ chanNum * PROCESSNUMCELLS ], PROCESSNUMCELLS , NULL );
……
}
```

然后对视频端口接收摄像头采集的数据进行处理，将 FIFO 缓冲区分成两个小的缓冲区，进行乒乓的处理模式可以提高程序的效率，一帧数据采集完之后便触发 DMA，重建过程中的解码视频图像将通过视频端口输出到显示器。

6.3.2　代码移植

代码的移植就是将在 PC 上能够运行的程序移植到 DSP 端，进行一定的调整，使其满足 DSP 的语法规范，能够初步跑通，这包括以下几部分工作。

1. 冗余代码删除

现有的程序结构复杂，冗余度很高，里面有许多和 DSP 端实现无关的代码，可以进行删除处理，以提高代码执行效率。例如，原始代码中有大量 debug 信息，trace 信息，assert 信息及 print 函数等，这些都是代码编写过程中调试所需要的信息，在实现到 DSP 端时，可以删除。此外原始代码中还有计算 SNR 的函数及大量的统计函数，运算量大，这部分也需要删除掉。

由于 DM642 板同 PC 通信是通过仿真器由 JTAG 口相连，数据传输速度较慢，编码后的码流传入 PC 存储也有相当大的时间消耗，在具体实现中，我们在板上开辟一片存储空间，存储压缩后的码流，待整个视频序列编码结束后，再统一将编码后的码流传送到 PC 存储，这样和 PC 通信的时间就不占用编码的总时间。

2. 内存空间分配

与 PC 平台上程序开发的流程不同，为了能够用 TI 提供的编译器对 C 代码进行编译，需要对程序运行时所涉到的数据段和代码段分配存储空间。理想情况是所有的代码和数据都放到速度最快的内部存储器（ISRAM）中，但因为受大小限制，H.264 编码器代码无法全部放在内部存储器中，因此只能把频繁使用的代码放入其中，其余代码放置在外部存储器（SDRAM）中。通常的分段设置如下（可通过 DSP/BIOS 进行配置或者直接修改 CMD 文件）。

1）初始化分段

.cinit：全局变量初始化值和常用表格。

```
.cinit:     {} > SDRAM
```

.text：可执行代码。

```
.text:      {} > SDRAM
```

.switch：大量 switch 语句表格。

```
.switch:    {} > SDRAM
```

.const：字符串、浮点常量及定义为 const 的数据。

```
.const:     {} > SDRAM
```

2）未初始化分段

.stack：存放局部变量及函数调用时保存现场和参数传递。

当程序在 DSP 上运行起来之后，代码要访问最频繁的变量往往是一些暂存在栈中的局部变量，为了使这些需要频繁访问的变量得到最快的访问速度，选择将 .stack 分段放置到 ISRAM 中。.stack 分段中除了局部变量以外，还有函数调用时放入栈中的参数、函数的返回地址、函数调用时保存的现场数据等。对于那些调用次数少的函数可以直接用函数体代替，或者用内联（inline）函数，这样减少由于函数调用增加的现场保护消耗，提高代码效率。

```
.stack: fill=0xc0ffee {
            GBL_stackbeg = .;
            *(.stack)
            GBL_stackend = GBL_stackbeg + 0x1000 - 1;
            _HWI_STKBOTTOM = GBL_stackbeg + 0x1000 - 4 & ~7;
            _HWI_STKTOP = GBL_stackbeg;
        } > ISRAM
```

.sysmem：动态内存分配的预留空间。

对于在 DSP 上运行的程序，我们将所有 C 代码中通过 malloc 动态申请的内存空间配置成静态空间。这主要基于两点考虑：第一，算法中所使用的视频数据缓冲区大小都比较固定，并且要被频繁访问，空间的利用率本身就很高；第二，动态地在堆（head）中分配和释放内存空间本身就需要消耗一定的 CPU 时间。在程序中，将需要动态分配的空间用静态数组代替，并将这些数组指定到自定义的代码段中。

.bss：全局和静态空间。

我们注意到，小存储器模式要求整个.bss 段的大小在 32 KB 的范围内，而大存储器模式对.bss 分段的大小不限制。由于算法经过编译后，全局和静态变量已经超过 32 KB 的大小，因此必须选择大存储器模式。

```
.bss:       {} > SDRAM
```

.far：声明为 far 类型的全局和静态变量。

```
.far:       {} > SDRAM
```

3）自定义分段

同时，在程序中也可根据需要指定所需变量与函数的内存分配，如将下述变量指定为 mydata 块。

```
#pragma DATA_SECTION(mb, ".mydata")
#pragma DATA_SECTION(analysis, ".mydata")
#pragma DATA_SECTION(dct, ".mydata")
```

对应的 CMD 文件中内存分配为：

```
SECTIONS
{
    .vecs                > ISRAM
    .mydata              > ISRAM
    .myconst             > ISRAM

    .sad_16x16           > ISRAM
    .sad_16x8            > ISRAM
……

    .macroblock_probe_pskip > ISRAM
.macroblock_encode_pskip > ISRAM
……

    .bs_write_se > ISRAM
    .bs_write_te > ISRAM
……

    .mc_xy33      >    ISRAM
    .mc_xy21      >    ISRAM
    .mc_xy12      >    ISRAM
    .mc_xy32      >    ISRAM
    .mc_xy23      >    ISRAM
}
```

以上这些分段均可以对 CMD 文件进行配置，根据程序的需要，将上述的段映射到不同的物理存储空间。CMD 文件是链接器命令文件，需要在该文件中说明系统的存储器配置情况及制定程序和数据的具体存放地址。

3. 代码的标准 C 化

TI 公司为 DSP 软件开发提供开发工具 Code Composer Studio（CCS）作为应用调试仿真的集成开发环境（IDE）。CCS 提供了基于标准 C 的编译器和优化器，通过 CCS 的 C 编译器优化器，标准 C 语言写成的代码就可以进行编译，在 DSP 上运行，因此，需要将 C 语言代码标准 C 化。将 PC 上运行代码移植到 DSP 上时，涉及以下几方面工作。

（1）去除原始代码中的文件操作，如文件读取/输出等。因为对于在开发板上运行的 DSP 程序来说，没有文件的概念，只有存储空间的概念。原始视频流通过仿真器或摄像头捕获到板上缓冲区，编码完后的码流也存放在内存中。

（2）规范数据类型，对于 DM642 来说，它是一款定点 DSP，支持的数据类型有限，标准 C 化时需要对数据类型进行规范。

（3）更改一些第三方厂商对标准 C 的扩展，因为原始程序使用 Visual Studio 开发，微软对相应代码进行了扩充，在实际的移植过程中需要进行一些关键字的修改和替换工作。

6.4　代 码 优 化

通过代码移植能够获得在 DSP 上初步运行的代码，但由于没有考虑到 DSP 自身的硬件特点，不适合 DSP 强大的并行处理能力，因此执行效率低下，不能满足实时要求，需要对其进行进一步优化。

对 DSP 代码的优化有三个层次，分别是项目级优化，算法级优化，指令级优化，其中算法级优化和 H.264 编码原理相关，不是本章讨论的重点，此处从略。

下面将分别介绍项目级优化和指令级优化的相关内容。

6.4.1　项目级优化

项目级优化是对项目的整体优化。首先，在对整个项目进行编译链接生成 DSP 代码时，合理选择配置编译器选项，并针对这些参数选择，对程序进行调整和修正。TI 提供的 C 语言编译器提供了若干等级和种类的自动优化选项，根据 TI 的文档说明，编译器优化的效果可以达到直接用汇编程序编写性能的 80%以上。与优化相关的选项介绍如下。

-ms0：不使用冗余循环进行优化，从而能够减小程序的大小，该选项可以在希望减小代码长度的情况下使用。

-o0：简化控制流，去除程序中未使用的代码，简化表达式并对内联函数调用进行扩展。

-o1：除了具有所有-o0 的功能外，还具有局部变量传播，去除未使用的赋值语句及去除局部共同表达式的能力。

-o2：除了具有所有-o1 的功能外，还具有安排软件流水对循环进行优化，去除全局共同表达式及全局未使用赋值的功能，同时还将把循环中的数组的引用转化为指针访问的形式，加快数组访问速度。

-o3：除了具有所有-o2 的优化功能外，还具有去除所有未调用函数，对小函数进行内联调用等功能。-o3 选项是在程序级别进行的最高程度的优化，一般编译优化到最后都要使用这个选项。但是有时候使用-o3 选项，程序无法正确运行，这往往是因为编译器对代码进行了错误的优化导致的。解决的方法是首先关闭-o3 选项，并打开-g 选项进行调试，调试通过后重新打开-o3 选项，直到没有错误为止。

-mt：告诉编译器程序中不存在多个指针同时指向同一块存储空间的情况，消除了存储区域的数据相关性，使得软件流水操作得以更好地运行，提高代码运行的并行性。

-mh[n]：去掉流水线的前序和结语，减小循环体内的代码长度，从而缩短循环体执行时

间。在某些时候，去掉流水线的前序和结语可能会出现读取地址超出有效范围的问题，所以要在数据段的开始和结语处增加一些填充，或者在分配内存时保证数组的前面和后面一段范围内都是有效地址。

-pm：指示编译器做程序级别的全局优化，将程序中所有文件联合在一起进行优化。

-mw：在汇编语言文件中加入软件流水信息，该选项对于流水线性能的分析非常重要。

-mln：编译生产大内存模式的程序。

-ml0：默认情况下将集合变量（数组和结构）声明为远类型。

-ml1：默认情况下将全部函数声明为远类型。

-ml2：等效于同时使用-ml0 和-ml1 选项。

-ml3：默认情况下将全部数据和函数声明为远类型。

在项目级优化进行的工作为项目编译时，通过参数-o3 调用最高级别的软件流水线优化，通过参数-mw 调用软件流水线循环反馈，使用-mt 消除代码块之间的相关性，从而增大程序的并行性。

示例 6-2 为在项目中实际使用的编译选项设置，该信息可从项目文件夹内的 log 文件中获得。

示例 6-2　H.264 编码：编译选项设置。

```
"c:\ti\c6000\cgtools\bin\cl6x" -gp -k -q -s -o3 -fr"./obj/"
-i"c:/ti/ddk/include"
-i"../include"
-i"c:/ti/referenceframeworks/include"
-i"c:/ti/boards/evmdm642/include" -d"_DEBUG" -d"_PAL"
-d"CHIP_DM642" -d"C6000" -mt -mw -mh -ml3 -mv6400
-@"../Debug.lkf" "msplab264main.c"
```

此外项目级优化时还应将只读变量声明成 const 型，告诉编译器该变量不会在程序运行中发生变化，从而提高 DSP 代码的并行性，如下面分析宏块信息的代码。

```
const short i_mb_y = mb_xy / h->sps->i_mb_width;
```

同时还应对程序结构进行调整，对不适合 DSP 执行的语句进行改写，以提高代码的并行性。例如，DSP 并行性很高，能够对代码进行流水线处理，但是原始代码存在大量条件判断语句，会对流水线造成中断，不利于代码的并行处理，因此，可以采用判断提前，去除不必要的判断等方式减少判断语句对流水线的中断。

6.4.2　指令级优化

1. 优化 C 语言代码

通过优化 C 语言代码，有助于 C 编译器产生高效率的汇编代码。主要包括以下几方面的内容。

1）数据类型的使用

C6000 编译器支持的数据类型有：字符型（char，8 位）、短整型（short，16 位）、整型

（int，32 位）、长整型（long，40 位）、浮点型（float，32 位）和双精度浮点型（double，64 位）。在选用数据类型的时候，最基本的原则是不使用超过实际运算所需最大的数据长度；另外，对于定点的乘法，最好选用短整型作为乘法器的输入，这样能够最有效地使用乘法器。对循环变量应当使用整型或者无符号整型（unsigned char），而不使用短整型或者无符号短整型，避免不必要的符号扩展。

2）消除数据和指令之间的相关性

为了使指令可以并行运行，编译器必须首先确定相邻指令之间的相关性，只有前后不相关的指令才能够并行处理，而用户可以用一些简单的关键字帮助编译器确定哪些指令不相关。比如，如果函数入口参数是指针参数，且函数内部没有其他指针指向相同的区域，则应该在函数声明中的入口指针前加上 restrict 关键字，告诉编译器该函数运行时不会产生指针重合的情况。对于执行过程中不会改变值的变量，可在其定义前加上 const 关键字，这样可以帮助编译器识别其内存依赖度，更好地对该代码段进行并行优化。如下面 msplab264_encoder_encode 函数的声明。

```
int        msplab264_encoder_encode( msplab264_t * restrict h,
msplab264_nal_t ** restrict pp_nal, int * restrict pi_nal,
msplab264_picture_t * restrict pic, msplab264_param_t * restrict
param, msplab264_out_t * restrict out, msplab264_mb_t * restrict
mb, msplab264_dct_t * restrict dct,
msplab264_mb_analysis_t * restrict analysis)
{
    ……
}
```

3）使用内联函数（intrinsics）

TMS320C64x 编译器提供的内联函数是直接映射为汇编指令操作的特殊函数。内联函数可以理解为已经经过汇编语言优化的 C 函数，它具有调用方便，同时又兼有高效性的特点。平均而言，一个内联函数单个时钟周期能够完成的操作往往需要普通的 C 函数十多个时钟周期才能完成。下面介绍一些媒体处理有关的函数。

add4：加法指令，一次执行 4 对 8 位数的加法。1 个寄存器有 32 位，可以存放 4 个 8 位数据。计算中，2 个源寄存器中的 4 组对应 8 位数据分别相加，结果存放在目标寄存器中。

```
int    _add4 ( int src1 , int src2 );
```

avgu4：一次执行 4 对 8 位无符号数据的平均运算。计算中，2 个源寄存器中的 4 组 8 位无符号型紧缩字求平均，结果以 4 个 8 位紧缩字的形式存放在目标寄存器中。

```
unsigned _avgu4 ( unsigned , unsigned );
```

dotpu4：一次执行 4 对 8 位无符号数据点乘运算。计算中，2 个源寄存器中的 4 组 8 位无符号型紧缩字对应相乘，乘积相加，所得结果存放在 32 位寄存器中。

```
unsigned _dotpu4 ( unsigned src1 , unsigned src2 );
```

subabs4：一次执行 4 对 8 位无符号数据求差绝对值运算。计算中，2 个源寄存器中的 4

组 8 位无符号型紧缩字对应相减，差值求绝对值，所得结果以 4 个 8 位紧缩字的形式存放在目标寄存器中。

```
int _subabs4 ( int src1 , int src2 );
```

ldb/ ldh/ ldw/ lddw：将 8 位、16 位、32 位或 64 位数据读入目标寄存器中，所读取的数据在内存中是地址 align（32 位对齐）的数据。

ldnw/ ldndw：将一个 32 位或 64 位的非对齐数据读入目标寄存器中。

stb/ sth/ stw/ stdw：将 8 位、16 位、32 位或 64 位数据写入内存中，所写入的数据在内存中是地址 align（32 位对齐）的数据。

stnw/ stndw：将一个 32 位或 64 位的非对齐数据写入内存中。

```
long long & _amem8 ( void * ptr );
```

下面以一段读取内存的子函数为例，说明内联函数的使用。

```
for ( j =0; j < BLOCK_SIZE ; j ++)
for ( i =0; i < BLOCK_SIZE ; i ++)
imgY[img->pix_y+block_y+j][ img->pix_x + block_x + i]=img->m7[j][i] ;
```

这段代码的作用是进行一个块的数据复制，读内存指令需要 4 个时钟周期完成，对于一个 4×4 块而言，估算复制 16 B 的数据所需要的时钟数为（4+1）×16=80 个。

内联函数_mem4(void *ptr)允许一次完成对 4 B 数据的读写操作，于是代码可以改写为：

```
for ( j=0; j < BLOCK_SIZE ; j ++)
{
data = mem4(&(img->m7[j][0]));
_mem4(&imgY[img->pix_y + block_y+j][img->pix_x + block_x +0])=data;
}
```

这个循环只需要（4+1）×4=20 个时钟周期，是原来时钟数的 25%。

在插值、运动估计、DCT 变换中有大量块数据复制，使用内联函数可以大大提高代码执行效率。

另外一个在实际 H.264 编码器中使用的例子是求块平均像素值。

示例 6-3　H.264 编码：内联函数的使用。

```
for( y = 0; y < height; y++ )
{
    for( x = 0; x < width; x+=4 )
    {
       _mem4(&dst[x])=_avgu4(_mem4_const(&dst[x]),_mem4_const(&src[x]));
    }
    dst += i_dst;
    src += i_src;
}
```

2. 改善软件流水

如果程序中判断跳转过多，或者程序的循环嵌套的深度过大，将严重影响 DSP 发挥其

并行性效果。另外，DSP 的并行性效果还反映在其对循环的软件流水分配上，但是，DSP 软件流水的限制很多，其中一条就是只对最内层循环做软件流水分配，这样循环嵌套太多，就不利于流水线的分工，所以需要对程序内部所有的两层或者三层循环嵌套进行拆解。

事实上，跳转类指令会损坏软件流水，每个跳转指令都有大概 5 个延迟间隙，这是一项很耗时的工作，应尽可能减少程序中的跳转分支。对于多重循环的控制，若外层循环较少，可将内层循环展开，把转移条件结合起来，以减少层与层之间的相互联系，具体可以考虑以下几点：

1）避免循环体内使用太多局部变量

如果循环的长度太长，中间的运算结果较多，使得需要寄存器的数量大于 64 个，将导致软件流水不能建立。因此最内层循环使用的变量数不能太多。如果内部寄存器数量不够，则可以将内层循环拆分成多个循环，建立多个软件流水线。

2）避免循环内调用函数

循环体中的函数调用会阻碍软件流水的建立，如有可能需要避免。例如，使用内联函数代替，或者将函数声明为 inline 类型，在编译阶段将函数代码在调用点展开。如示例 6-4 所示。

示例 6-4　H.264 编码：inline 函数的使用。

```
static inline void msplab264_reference_build_list( msplab264_t
* restrict h, int i_poc, msplab264_param_t * restrict param )
```

3）关键代码放置在最内层循环

如果循环体内包含多层循环，则需要将耗费最多时钟周期的关键代码放置在最内层循环，否则达不到软件流水优化的目的。如有可能，需要手动修改循环体，将包含关键代码的所有内层循环进行展开，使编译器可以对最内层循环建立流水。

4）避免循环体内包含条件判断

这种情况可能需要将条件判断转移到循环体外。

6.4.3　缓存优化

把缓存优化与其他优化方法单独列出来，是由于缓存优化与其他方法有一定区别，需要更了解 DM642 的底层体系结构。

缓存（cache）优化的性能依赖于缓存行（line）的重复使用。如果缓存中的数据不常用，则很可能被外存中其他位置的数据取代。访问不在缓存中的内存行会导致 CPU 的延迟。只要该行在缓存中，接下来对该行的访问将不会引起延迟。因此，在该行被取代出缓存之前，需要尽可能多地重复使用。

一般而言，主要有以下两种途径来进行缓存优化。

（1）减少缓存不中（miss）的数量。所谓缓存不中，即访问的行不在缓存中。这可以通过如下办法达到：

最大化缓存行的重复使用，在一个缓存行中访问所有的内存位置，这是由于数据分配到缓存中会引起比较多的延时。在同一个缓存行的相同的内存位置应当尽可能多被重复使用，这样该数据被重复读取的时候不会发生缓存不中。

只要数据还可能被继续访问，该行应该避免被驱逐。内存中的数据如果在访问时没有其他的对应行数据被访问，则不会被其他行驱逐。

（2）对于每次缓存不中需要减少延迟周期的个数，这要通过流水线来达到目的。

1. 应用级优化

选择合适的 L2 缓存大小是非常重要的。一般而言，至少需要 32 KB 的缓存。

对于通过 EDMA 输入的数据流，需要把数据分配到 L2 SRAM 中，这与把数据存放到外存相比有以下几个优点：

（1）L2 SRAM 更靠近 CPU，因此，通过这种方式可以减少延迟。

（2）缓存一致性通过缓存控制器自动保持，无需用户的操作。如果 buffer 放置在外存中，需要小心地考虑 L2 缓存一致性的问题。

（3）额外的一致性操作可能会增加到延迟中。

在应用中，需要对 DSP 类处理指令和一般目的处理指令进行区分，DSP 类处理指令相对于一般目的处理指令具有更好的优化性能，一般处理指令主要包括控制流和条件跳转指令，这些指令不具备很好的并行性能，在执行的过程中依赖于很多模式及条件，充满了不可预见性，即数据内存访问是非常随机的，这导致优化过程非常困难，因此，在 L2 SRAM 不足以包含整个应用程序的代码和数据时，最好将一般目的代码放在外存中，通过 L2 缓存来处理内存访问，这样就可以使 L2 SRAM 更有效地处理最重要的信号处理指令。由于一般目的指令的不可预测性，L2 缓存需要设置为相对较大。对于 64x 系统，L2 缓存的结合数（set-associativity）为 4，因此可以根据需要在 32～256 KB 之间配置，这里初步选择 128 KB 的 L2 缓存。

2. 程序级优化

程序级优化主要考虑如何编排存储空间中的代码和数据，如何改变函数调用方式。对于单个函数是没有必要改动的，需要优化的是函数所访问的数据结构。通过数据结构的调整来使缓存优化最大化。为了提高数据重复利用率，我们细致了解了 H.264 编码器所涉及的数据结构，并对数据结构进行重新改写，将最常用的数据放到 ISRAM 中（见 6.3.2 节的自定义分段），这包括查找表等频繁使用的数据。

通常，最终目标是减少缓存不中的数目及因此带来的延迟周期。缓存不中的减少可以通过减少内存的容量并重复使用已有的缓存行来实现。重复使用可以通过避免驱逐及写到预分配的行中实现，而延迟周期的解决办法是利用流水线。

读不中主要有三种情况，下面分别解释并阐述如何解决。

（1）所有的数据和代码都能够放置到缓存中，但是发生了冲突不中，这种不中主要是由于代码的存储空间占据了同一个缓存行造成的，需要把代码和数据在内存中连续分配。

实际上，编译器和连接器不会考虑冲突的问题，只有不合适的内存分配才会导致程序执

行中发生冲突。考虑一段简单的代码：

```
for ( i =0; i<N; i ++)
{
function1();
function2();
}
```

假如 function 1 和 function 2 被连接器安排到的存储空间对 L1 代码缓冲区是重叠的，那么且不说第一次调用 function 1 和 function 2 必然造成不中，以后调用 function 1 将会驱逐 function 2，这样将导致每次迭代都会发生 L1 程序缓冲区不中。

对于这种情况，必须通过安排两个函数到相对 L1 代码缓冲区不重叠的存储空间中，最直接的方法就是把两个函数放置到相邻的存储空间。具体的操作可以考虑以下两种方法：一种是用编译选项-mo 把每个函数都放到独自的段中，通过编译器检查 map 文件；二是用伪指令 CODE SECTION 进行定义。

对于 L1 数据段冲突不中的问题，与 L1 代码缓冲区的问题是类似的，区别在于 L1 数据缓冲区的缓存是两路的结合数，而 L1 代码缓冲区是直接映射的模式。

（2）数据集大于缓存且连续存储，但是无法重复使用，发生冲突不中，这种情况下可以通过交织的手段来减少缓存不中。

在这种读不中的情况下，由于数据集大于缓存，因此即使数据集是连续存放到存储空间中，数据是不能重复使用的。发生这种不中主要是由于数据访问之前，至少有两个读不中发生在同一行。这种情况下，需要在存储空间对数据集进行连续排列，并引入一些填充来实现数据交织。

以一个简单的点乘算法为例来说明这个过程，程序如下所示。

```
int w_dotprod ( short    *w, short    *x , short    *h , int N)
{
int i , sum = 0 ;
for ( i =0; i<N; i ++)
sum += w[ i ] * x [ i ] * h [ i ] ;
return sum;
}
```

假设三个数组 w[]，x[] 和 h[] 都放置在存储空间中对于 L1 代码缓冲区的同一位置，将会出现 L1 代码缓冲区的颠簸问题。在循环的第一次迭代中，三个数组都发生了读不中，而第三个读不中则将前两个读不中引入的行驱逐出去。假定在第一次迭代后没有重新分配到相同位置，那么 w[1]和 x[1]在下一周期访问时便能够缓冲命中。然而，由于 h[0]的访问导致 w[0]所在行被驱逐，将带来 w[1]不中问题，这样，在数组的每个数据被重复使用之前都会被驱逐，就发生了冲突不中，因此，必须进行填充实现数据交织，从而避免上述问题。

（3）数据集大于缓存且冲突不中和容量不中（capacity miss）都发生（由于相同数据发生重复使用），此时可以将数据集分解并在一个时段只处理一部分数据。

在这样读不中的情况下，数据重复使用，但由于数据集超过了缓存容量，导致容量不中和冲突不中。下面以一个简单的点乘算法为例说明这个过程，程序如下所示。

```
short in1[N] ;
short in2[N] ;
short in3[N] ;
short in4[N] ;
short w[N] ;
r1 = dotprod ( in1 , w, N) ;
r2 = dotprod ( in2 , w, N) ;
r3 = dotprod ( in3 , w, N) ;
r4 = dotprod ( in4 , w, N) ;
```

假定每个数组都是 L1 数据缓冲区的两倍容量，在第一次调用时，in1[] 和 w[] 都不中，而在接下来的调用中，希望 w[] 能重复使用。然而，每次调用完函数之后，w[] 起始点都被最后的数所取代，这是由于容量不足造成的。

在缓存容量用完的时候，w[] 的第一行必然首先被驱逐。在本例中，缓存容量在 N=4 个输出计算之后就已经用完。如果在这个时候停止处理 in1[] 而开始处理 in2[]，那么能够重复使用刚刚配置在缓存中的 w[]元素。此后，在计算完下一个 N=4 个输出之后，跳过处理 in3[] 而处理 in4[]。通过这种手段，就能在下一个 N=4 输出时处理 in1[]，如此重复。

重新构成的代码如下所示。

```
for ( i =0; i <4; i ++)
{
o=i* N/4 ;
dotprod(in1+o , w+o , N/4 ) ;
dotprod(in2+o , w+o , N/4 ) ;
dotprod(in3+o , w+o , N/4 ) ;
dotprod(in4+o , w+o , N/4 ) ;
}
```

进一步，可以通过 touch loop 在迭代开始之前对数据集进行分配。值得注意的是 LRU 策略只要同样行的两路按相同顺序访问，就能自动保持行使之命中。如果访问顺序改变，LRU 行为便得不到保证。

6.4.4 优化结果

通过以上的优化手段，目前编码器的效果是对 CIF（352×288）序列，中等运动下能够超过实时编码。对于 D1（704×576）序列，中等运动下，每秒钟编码 10 帧以上，即达到准实时。

6.5 程 序 示 例

该示例的 DSP/BIOS 配置文件如图 6.5 所示。

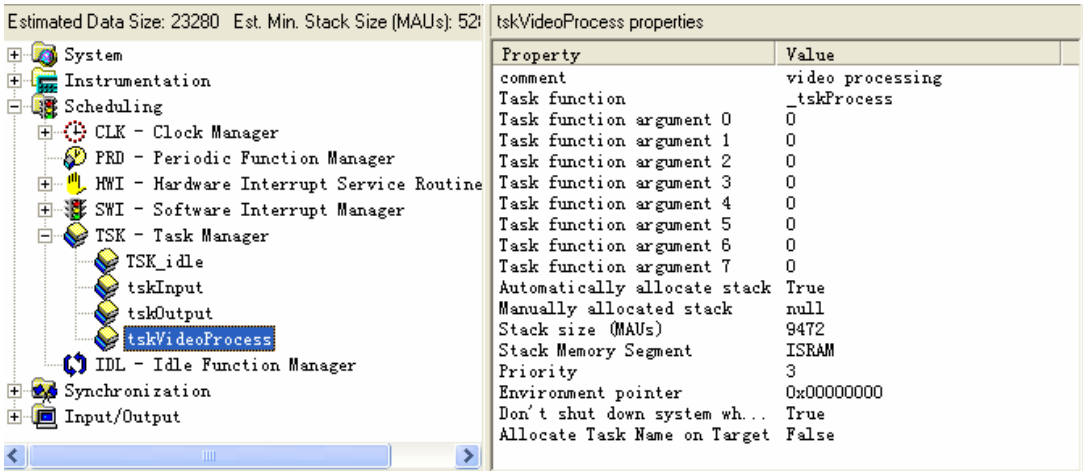


图 6.5　DSP/BIOS 配置文件

整个系统由 4 个任务组成，TSK_idle，tskInput（输入），tskOutput（输出）和 tskVideoProcess（实际编码过程）。其中 TSK_idle 为守护进程，用于传递处理器的信息，tskInput 和 tskOutput 为输入、输出模块，在这里不再详细介绍。这里主要列出系统初始化的 main 函数和处理编码的 tskVideoProcess 任务函数 tskProcess 代码，以供参考。

初始化函数 main（）：

```
void main()
{
    int i;
    CSL_init();
    CACHE_clean(CACHE_L2ALL, 0, 0);
    CACHE_setL2Mode(CACHE_128KCACHE);
    CACHE_enableCaching(CACHE_EMIFA_CE00);
    CACHE_enableCaching(CACHE_EMIFA_CE01);
    CACHE_enableCaching(CACHE_EMIFA_CE02);
    CACHE_enableCaching(CACHE_EMIFA_CE03);
    CACHE_enableCaching(CACHE_EMIFA_CE04);
    CACHE_enableCaching(CACHE_EMIFA_CE05);
    CACHE_enableCaching(CACHE_EMIFA_CE06);
    CACHE_enableCaching(CACHE_EMIFA_CE07);
    #define L2ALLOC3 0x0184200C
    *(int *) L2ALLOC3 = (*(int *) L2ALLOC3) | 0x00000007;
    #define L2ALLOC2 0x01842008
    *(int *) L2ALLOC2 = (*(int *) L2ALLOC2) | 0x00000007;
    #define L2ALLOC1 0x01842004
    *(int *) L2ALLOC1 = (*(int *) L2ALLOC1) | 0x00000007;
    CACHE_setPriL2Req(CACHE_L2PRIHIGH);

    /*----------------------------------------------------*/
```

```
/*Initialize RF5 modules                                          */
/*-------------------------------------------------*/
CHAN_init();
ICC_init();
SCOM_init();

/*-------------------------------------------------*/
/*Setup the RF-5 chan module                                      */
/*-------------------------------------------------*/
CHAN_setup( intHeap, extHeap, intHeap, 1, NULL, NULL);

/*-------------------------------------------------*/
/*Initialize and start the process task                 */
/*-------------------------------------------------*/
tskVideoInputInit();
tskVideoOutputInit();

/*-------------------------------------------------*/
/*Initialize and start the Input task                   */
/*-------------------------------------------------*/
tskVideoInputStart();
tskVideoOutputStart();

/*-------------------------------------------------*/
/*Initialize and start the process task                 */
/*-------------------------------------------------*/
tskProcessInit();

/*-------------------------------------------------*/
/* create all SCOM and message objects                   */
/*-------------------------------------------------*/
for(i = 0; i<NUM_SCOM_OBJS_IN; i++) {
        objSCOMInToProc[i][0] = SCOM_create("INTOPROC", NULL);
        objSCOMInToProc[i][1] = SCOM_create("PROCTOIN", NULL);
}

for(i = 0; i<NUM_SCOM_OBJS_OUT; i++) {
        objSCOMProcToOut[i][0] = SCOM_create("PROCTOOUT", NULL);
        objSCOMProcToOut[i][1] = SCOM_create("OUTTOIN", NULL);
}

for(i = 0 ; i < NUM_OF_CAPTURE_FRAMES; i++)
{
```

```
            YUV_buffer_for_capture[i][0] =
                YUV_buffer_for_capture_all    +
                (i * ((DEMO_WIDTH * DEMO_HEIGHT * 3) /2)) ;
            YUV_buffer_for_capture[i][1] =
                YUV_buffer_for_capture[i][0] +
                (DEMO_WIDTH * DEMO_HEIGHT) ;
            YUV_buffer_for_capture[i][2] =
                YUV_buffer_for_capture[i][1] +
                ((DEMO_WIDTH * DEMO_HEIGHT)/4 );
        }
        capfrnum = 0;
    }
```

系统执行完 main 函数之后，进入 DSP/BIOS 控制，由图 6.5 所示的三个任务负责输入、输出和处理，其中涉及的实际视频编码函数为 tskProcess，代码如下：

```
    /*-------------------------------------------------------*/
    /* The task will handle the processing part :            */
    /* -Will get the message from the Input task with input  */
    /*    frame pointers                                     */
    /* -Will execute the channel to encode and reconstruct   */
    /* -Will pass the decoded frame pointers to output task  */
    /*-------------------------------------------------------*/
    void tskProcess()
    {
        unsigned int *bufs;
        ScomBufChannels *pMsgBuf;
        SCOM_Handle fromInputtoProc,fromProctoOut;
        fromInputtoProc = SCOM_open("INTOPROC");
        fromProctoOut      = SCOM_open("PROCTOOUT");

        if( ( h = msplab264_encoder_open( param, out ) ) == NULL )     //初始化 h 结构体
        {
        }
            msplab264_macroblock_cache_init( h, &mb );
            pic = msplab264_picture_new( h, param );   //初始化 pic  结构体，为 Y,U,V 分配空间
        while(1)
        {
      /*-------------------------------------------------      */
    /* Wait for the message from input task to recieve captured */
      /* frame to be cycled through encoding and decoding.      */
    /*-------------------------------------------------------*/
            pMsgBuf = SCOM_getMsg(fromInputtoProc, SYS_FOREVER);
            bufs = pMsgBuf->bufChannel;
```

```
                yuvBufIp[0] = (char *)bufs[0];
                yuvBufIp[1] = (char *)bufs[1];
                yuvBufIp[2] = (char *)bufs[2];
                pic->plane[0] = (unsigned char *)yuvBufIp[0];
                pic->plane[1] = (unsigned char *)yuvBufIp[1];
                pic->plane[2] = (unsigned char *)yuvBufIp[2];
                if( msplab264_encoder_encode( h, &nal, &i_nal, pic, param, out, &mb, &dct, &analysis)
< 0 )
                {
                }
                yuvBuf[0] = h->fdec->plane[0] - 16 ;
                yuvBuf[1] = h->fdec->plane[1] - 8;
                yuvBuf[2] = h->fdec->plane[2] - 8;
        /*-------------------------------------------------*/
        /* Send message to output task with pointers to decoded frame*/
        /*-------------------------------------------------*/
                thrProcess.scomMsgTx.status = 1;
                thrProcess.scomMsgTx.bufChannel= (void *)&yuvBuf[0];
                SCOM_putMsg(fromProctoOut,&(thrProcess.scomMsgTx));
                framenum++;
            }
            msplab264_picture_delete( pic );
            msplab264_encoder_close( h, param );
        }
```

在上述函数中调用了 msplab264_encoder_encode 函数，其代码如下：

```
        /************************************************************
          * msplab264_encoder_encode:
          *    XXX: i_poc     : is the poc of the current given picture
          *           i_frame : is the number of the frame being coded
          *    ex:   type frame poc
          *           I          0     2*0
          *           P          1     2*3
          *           B          2     2*1
          *           B          3     2*2
          *           P          4     2*6
          *           B          5     2*4
          *           B          6     2*5
        ************************************************************/
        int         msplab264_encoder_encode( msplab264_t * restrict h,
                    msplab264_nal_t ** restrict pp_nal, int * restrict pi_nal,
                    msplab264_picture_t * restrict pic, msplab264_param_t * restrict param,
                    msplab264_out_t * restrict out, msplab264_mb_t * restrict mb, msplab264_dct_t * restrict
dct,
```

```
                msplab264_mb_analysis_t * restrict analysis
                )
        {
            int        i_nal_type;
            int        i_nal_ref_idc;
            int        i_slice_type;
            int     i_global_qp;
            /* no data out */
            *pi_nal = 0;
            *pp_nal = NULL;
            /* ---------- Select slice type and frame ---------- */
            if( h->i_frame % (param->i_iframe * param->i_idrframe) == 0 )
        //处理 IDR 帧的情况，h->param.i_iframe * h->param.i_idrframe 为 IDR 的周期数
            {
                i_nal_type       = NAL_SLICE_IDR;
                i_nal_ref_idc = NAL_PRIORITY_HIGHEST;
                i_slice_type = SLICE_TYPE_I;
                /* we encode the given frame */
                h->picture       = pic;
                /* null poc */
                h->i_poc             = 0;
                h->fdec->i_poc = 0;
                /* reset ref pictures */
                msplab264_reference_reset( h, param );
            }
            else
            {
                /* TODO detect scene changes and switch to I slice */
                {
                    i_slice_type   =   h->i_frame   %   param->i_iframe   ==   0   ?   SLICE_TYPE_I   :
SLICE_TYPE_P;
                    /* we encode the given frame */
                    h->picture         = pic;
                    h->fdec->i_poc = h->i_poc;    /* low delay */
                }
                i_nal_type       = NAL_SLICE;
                if( i_slice_type == SLICE_TYPE_I || i_slice_type == SLICE_TYPE_P )
                {
                    i_nal_ref_idc = NAL_PRIORITY_HIGH;    /* Not completely true but for now it is
(as all I/P are kept as ref)*/
                }
                else /* if( i_slice_type == SLICE_TYPE_B ) */
                {
```

```
                i_nal_ref_idc = NAL_PRIORITY_DISPOSABLE;
        }
}
/* increase poc */
h->i_poc += 2;
if( h->picture == NULL )
{
        /* waiting for filling bframe buffer */
        return 0;
}

i_global_qp = param->i_qp_constant;

/* build ref list 0/1 */
msplab264_reference_build_list( h, h->fdec->i_poc, param );

/* --------- Create slice header    ------------- */
msplab264_slice_init( h, i_nal_type, i_slice_type, i_global_qp, param );

/* ---------- Write the bitstream ------------- */
/* Init bitstream context */
out->i_nal = 0;
bs_init( &out->bs, out->p_bitstream, out->i_bitstream );

/* Write SPS and PPS */
if( i_nal_type == NAL_SLICE_IDR )
{
        /* generate sequence parameters */
        msplab264_nal_start( h, NAL_SPS, NAL_PRIORITY_HIGHEST, out );
        msplab264_sps_write( &out->bs, h->sps );
        msplab264_nal_end( h, out );
        /* generate picture parameters */
        msplab264_nal_start( h, NAL_PPS, NAL_PRIORITY_HIGHEST, out );
        msplab264_pps_write( &out->bs, h->pps );
        msplab264_nal_end( h, out );
}
/* Write the slice */
msplab264_slice_write( h, i_nal_type, i_nal_ref_idc, param, out, mb, dct, analysis );

/* End bitstream, set output    */
*pi_nal = out->i_nal;
*pp_nal = &out->nal[0];
```

```
        /* -------- Update encoder state ------- */
        /* update cabac */
        /* handle references */
        if( i_nal_ref_idc != NAL_PRIORITY_DISPOSABLE )
        {
            msplab264_reference_update( h, param, mb );
        }
        /* increase frame count */
        h->i_frame++;
        return 0;
    }
```

其余部分的代码这里不再详细介绍。

参 考 文 献

［1］范嘉略．DM642 处理器的 H.264 编码器实现研究．多媒体信号处理实验室技术报告．清华大学电子工程系，2007.

第 7 章　使用 CCS 开发视频图像增强算法

图像增强（Image Enhancement）的首要目的在于处理图像使其比原始图像更适合于特定应用，例如，将原来不清晰的图像变得清晰或强调某些感兴趣的特征，抑制不感兴趣的特征等。常用的图像增强方法可分成频率域方法和空间域方法两种，前者把图像看成一种二维信号，因而可以通过傅里叶变换等方法对其在频率域进行处理，如采用低通滤波法去除噪声干扰，采用小波变换获取图像边缘信息等；后者的空间域是指处理的领域局限于图像平面自身，具有代表性的算法有直方图均衡化和中值滤波等，这些方法被广泛应用于低照度图像增强与降噪等领域。值得注意的是图像增强的通用评价标准是不存在的，需要由观察者针对特定的应用场景对增强的结果进行分析评估。

本章的内容包括以下几部分：7.1 节介绍用于低照度图像增强的直方图均衡化方法，7.2 节给出一个用 C 语言实现的示例程序，7.3 节主要介绍如何通过 Code Composer Studio（CCS）和 XDS560 仿真器的相关特性进行 DSP 程序调试，7.4 节讨论性能评估与程序优化的相关问题。

7.1　直方图均衡化的基本原理

直方图是图像的一种统计表达，对一幅灰度图像，其灰度统计直方图反映了该图中不同灰度级出现的统计情况。一幅图像及其灰度统计直方图如图 7.1 所示，横轴表示图像中各个灰度级像素的个数。

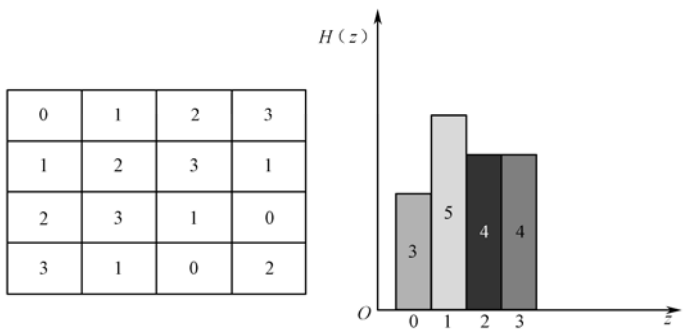


图 7.1　灰度图像及其直方图

严格地说，图像的灰度统计直方图是一个离散函数，可写成：

$$h(k) = n_k,\ k = 0,1,\cdots,L-1 \tag{7.1}$$

式（7.1）中 n_k 是二维图像 $f(x,y)$（这里将图像看成一个二维函数）中具有灰度值 k 的像素的个数。直方图的每一列（称为 bin）的高度对应 n_k。直方图提供原图中各种灰度值分布

的情况，也可以说给出了一幅图像所有灰度值的整体描述。

图像的视觉效果和其直方图有对应关系，或者说，直方图的形状和改变对图像有很大影响。直方图均衡化主要用于增强动态范围偏小的图像的反差。这个方法的基本思想是把原始图的直方图变换为均匀分布的形式，这样就增加了像素灰度值的动态范围，从而达到增强图像整体对比度的效果。其基本原理如下所述。

将公式（7.1）写成更一般的（归一化的）概率表达形式：

$$p_s(s_k) = n_k / n, 0 \leqslant s_k \leqslant 1, k = 0, 1, \cdots, L-1 \tag{7.2}$$

通过用图像里像素总个数进行归一化，直方图各列表达了各个灰度值像素在图像中所占的比例。图像增强实际可以看做一个灰度映射的过程，映射函数的定义域是原始图像像素的灰度值，值域是映射后图像像素的灰度值。其映射函数要满足两个条件：一是单调递增；二是对于 0 到 1 的定义域有 0 到 1 的值域。

不难发现，累计分布函数（Cumulative Distribution Function，CDF）满足上述两个条件，并有将原始图像灰度分布通过灰度映射变为均匀分布的效果。事实上 CDF 就是原始图的累积直方图，映射函数即为：

$$f(k) = \sum_{i=0}^{i=k} p_s(s_i), 0 \leqslant s_k \leqslant 1, k = 0, 1, \cdots L-1 \tag{7.3}$$

图 7.2 给出直方图均衡化的一个实例。原始图像较暗且动态范围较小，反映在直方图上就是其直方图所占据的灰度值范围比较窄且集中在低灰度值一边，而进行直方图均衡合后可以发现，增强图像的直方图占据了整个图像灰度值允许的范围。由于直方图均衡化增加了图像灰度动态范围，所以也增加了图像的对比度，反映在图像上就是图像有较大的反差，许多细节看得比较清晰了，但需要注意，直方图均衡化在增强反差的同时也增加了图像的可视粒度。

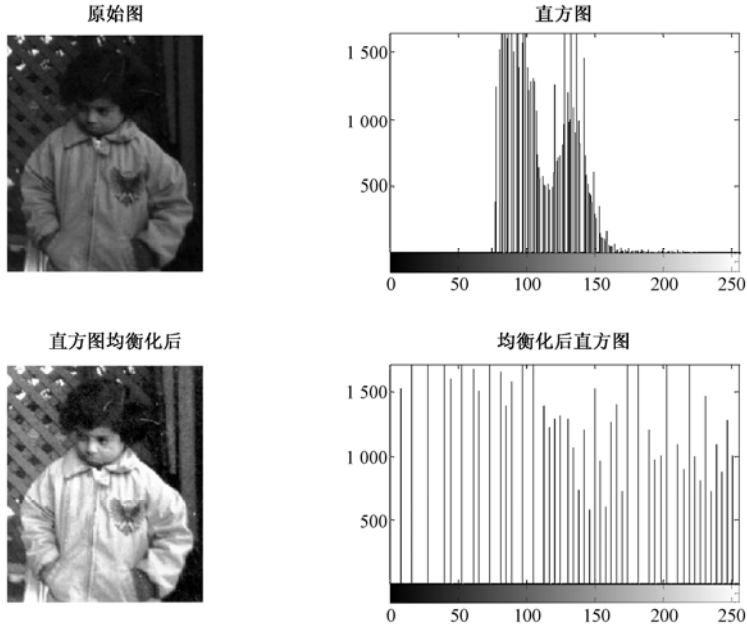


图 7.2　直方图均衡化实例

7.2　实 现 代 码

这一节将给出一个通过对视频图像每一帧进行直方图均衡化来实现视频图像增强的示例。因为这里是对每一帧图像进行处理，所以具体参数的获得及图像的采集与显示与 1.4.7 节无异，图像增强算法的实现主要集中在对最终获取的每帧图像（即这里的的 disFrameBuf）处理上，以下是直方图均衡的实现代码。

```
%001 void tskVideoLoopback()
%002 {
%003   ……
%004   Int i,j;
%005   Int currentPoint;
%006   Int nk[256],sk[256];
%007   char trans[256];
%008   ……
%009   while(1){
%010       /* copy data from capture buffer to display buffer */
%011       /*************************************************/
%012       for(i = 0; i < numLines; i ++) {
%013           DAT_copy(capFrameBuf->frame.iFrm.y1 + i *
capLinePitch,disFrameBuf->frame.iFrm.y1 + i * disLinePitch,numPixels);
%014       }
%015       DAT_wait(DAT_XFRID_WAITALL);
%016       for(i=0;i<256;i++)
%017       {
%018           nk[i]=0;
%019           sk[i]=0;
%020           trans[i]=0;
%021       }
%022       for(i = 0; i < numLines; i ++) {
%023           for(j= 0;j<disLinePitch;j++)
%024           {
%025               currentPoint=*(disFrameBuf->frame.iFrm.y1 + i *
disLinePitch+j);
%026               if(currentPoint<0)
%027                   currentPoint=currentPoint+256;
%028               nk[currentPoint]=nk[currentPoint]+1;
%029           }
%030       }
%031       sk[0]=nk[0];
%032       trans[0]=(char)((double)sk[0]/(numLines*disLinePitch)*255);
%033       for(i=1;i<256;i++)
%034       {
```

```
      %035          sk[i]=sk[i-1]+nk[i];
      %036
trans[i]=(char)((double)sk[i]/(numLines*disLinePitch)*255);
      %037       }
      %038       for(i = 0; i < numLines; i ++) {
      %039           for(j= 0;j<disLinePitch;j++)
      %040           {
      %041           currentPoint=*(disFrameBuf->frame.iFrm.y1 + i *
      disLinePitch+j);
      %042               if(currentPoint<0)
      %043                   currentPoint=currentPoint+256;
      %044               *(disFrameBuf->frame.iFrm.y1 + i
      *disLinePitch+j)=trans[currentPoint];
      %045           }
      %046       }
      %047       CACHE_clean(CACHE_L2ALL,0,0);
      %048       FVID_exchange(capChan, &capFrameBuf);
      %049       FVID_exchange(disChan, &disFrameBuf);
      %050   }
      %051 }
```

代码本身并不复杂，不再详细讲解，需要注意以下 3 点：

（1）任务处理函数（tskVideoLoopback()）的变量和循环数目相对与第 1 章的示例有所增加，要求给该任务分配更大的堆（stack），该项可以在 DSP/BIOS 的配置文件中进行修改。

（2）无符号整数字符向有符号类型转换的时候需要注意符号变化的问题，见代码的 026～027 行。

（3）因为这里使用了 DMA 操作，必须考虑数据一致性的问题，需要在两次数据传输的过程中清除 L2 缓存，因此第 047 行的 CACHE_clean 函数必须保留。

7.3　调　　试

Code Composer Studio 作为 TI 公司开发的一款优秀的集成开发环境，具有强大的调试功能。除了常规的断点、内存、寄存器观察功能外，还提供了几项在视频算法开发过程中非常有用的调试工具，这也是本节要讨论的重点。

7.3.1　DSP/BIOS 错误调试

在嵌入式平台开发的过程中，经常遇到的一个问题就是内存不足或者内存区域分配错误，在这种情况下程序能够正常编译、烧录，但在实际运行时会出现溢出情况（跑飞）。更糟糕的是这种错误是极其难调试的，它不具有可重复性，无法有效重现，而且出错的地方很难准确定位，这对构建复杂系统无疑是一个相当严峻的挑战。幸运的是 Code Composer Studio 针对 DSP/BIOS 开发专门提供了一系列监控系统当前运行状态的工具，下面以其中的

Kernel/Object View 为例进行说明（DSP/BIOS→Kernel/Object View）。

图 7.3 是一次在程序“跑飞”以后 Kernel/Object View 所显示的信息，从中可以发现，错误出现在 TSK 模块中，并且很有可能是 Stack 的大小问题（Stack Full），因此，一个可行的解决方法就是将 tskLoopback 的 Stack 增大，这里将它由 1 024（0×400）调整为 4 096（0×1 000），再次编译运行（在 DSP/BIOS 属性设置窗口中进行修改，如图 7.4 所示），结果正常，此时 Kernel/Object View 如图 7.5 所示，注意其中 Stack 的大小和 Stack Peak 的值。

图 7.3　Kernel/Object View 的出错信息

图 7.4　修改 tskLoopback 的 Stack 大小

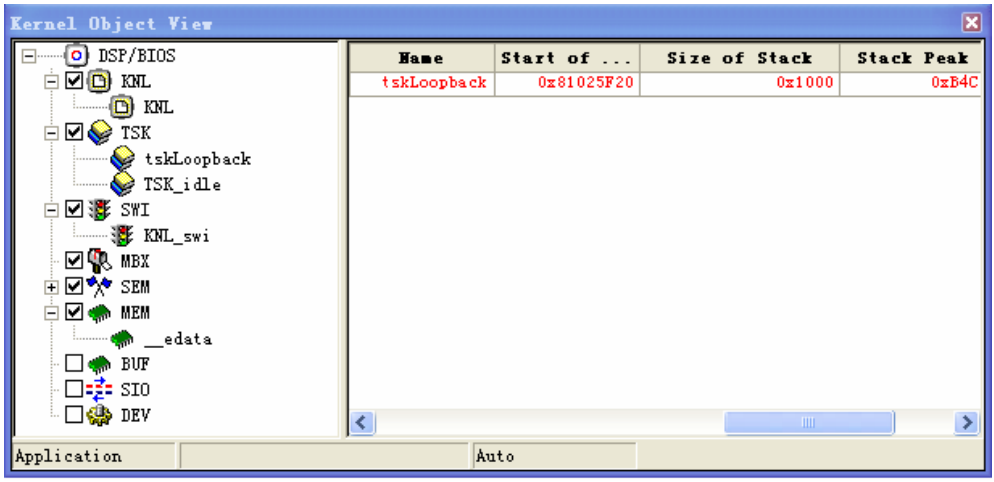


图 7.5　修改后 Kernel/Object View 的信息

7.3.2　使用 LOG 模块输出信息

输出信息管理（LOG-Event Log　Manager）是 DSP/BIOS 中一组可自定义的模块，它可以记录系统的日志信息并传回主机，结果通过 Message Log 工具进行捕捉。根据文档 TMS320 DSP/BIOS User's Guide（SPRU423F）的建议，在 DSP 开发过程中，应该尽可能所述。使用 LOG_printf 而不是一般 C 语言中常用的 printf 函数来输出信息，原因如下所述。

（1）LOG_printf 具有更快的速度，在 C6000 系列中 LOG_printf 函数和 LOG_event 函数只需 32 条指令即可完成。

（2）在 DSP 的实时运行支持库（RTS）中 printf 是通过断点实现，因为其相对于其他任务具有更高的优先级，因此，如果在代码中间使用 prinf 的话会影响整个 DSP/BIOS 的 RTDX，带来不希望的负效应。

如果要使用 LOG 模块，必须首先在 DSP/BIOS 中进行配置，配置选项中包括缓冲区长度、存储位置等，如图 7.6 所示的 trace 模块。而具体的使用方法与 printf 函数类似（注意：输出长度不能超过缓冲区大小），如在程序中可以通过以下语句调用：

```
LOG_printf(&trace, "Current Frame: %d",frames);
```

在 CCS 中打开 Message Log 即可接收 LOG_printf 输出的信息，如图 7.7 所示。

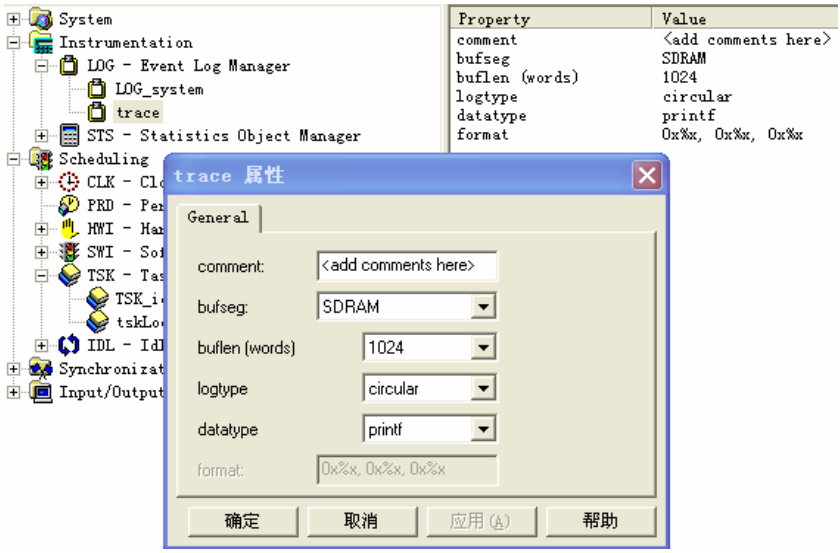


图 7.6　配置 Event Log Manager

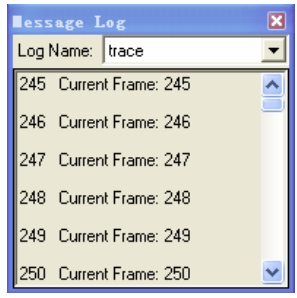


图 7.7　Message Log 对话框

7.4　算法性能优化

我们在编写视频处理算法的时候不仅关心算法的效果，更关心该算法能否在开发平台上实时高效地实现，因此这一节主要讨论两个问题：如何评估一个 DSP 算法的性能及如何优化。

7.4.1　如何评估一个 DSP 算法的性能

评估一个视频处理算法性能的最直观方法便是看该算法能否实现实时输出，直观地说就是是否有丢帧现象，但仅仅如此是不够的，因为在嵌入式平台必须考虑电耗的问题，我们期望算法能够在尽可能低的占用处理器时间的条件下完成任务。此外代码优化的同时也需要某种方式来获得系统各部分所消耗的 CPU 运算，以便有目的地进行优化。出于以上两个目的，性能评估工具在算法优化过程中有着极其重要的作用，其中比较常用的工具有以下几种：

1）Profile 工具

Profile 是 CCS 所提供的一套完整的系统分析与优化工具（见图 7.8），它不仅能够分析程序中每段代码所消耗的 CPU 周期，缓冲冲突情况，还提供了 Tuning Advice 和 Compiler Consultant 功能，可以让一个编程新手在不了解处理器结构的条件下就可以编写出相当高效的代码。

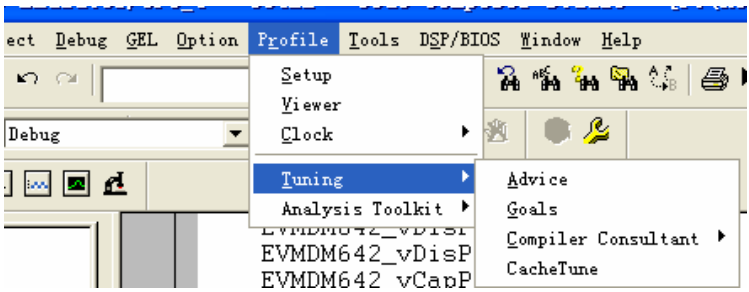


图 7.8　Code Composer Studio: Profile 工具

关于 Profile 工具的使用方法这里不详细介绍，需要注意的是它的大部分功能要求使用模拟器（Simulation，如 DM642_sim）而不是仿真器（Emulation）进行，因此很大程度上限制了它在实物仿真（如 EVM DM642+XDS 560）上的使用。

2）DSP/BIO 工具

在实际开发过程中，另外一个经常使用的是 DSP/BIOS 的相关工具（见图 7.9）。它通过一系列模块，如 LOG，STS 等，将 CPU 运行的信息通过 JTAG 仿真口传递回主机，进而可以对当前的工作状态进行分析。

其中 Kernel/Object View 和 Message Log 已经分别在 7.3.1 节与 7.3.2 节中进行过介绍，而这里对系统性能评估比较重要的是 Execution Graph（见图 7.10）和 CPU Load Graph（见图 7.11）。其中 Execution Graph 用来显示系统的时间片分配状态，而 CPU Load Graph 则可以直观地表示系统当前的负载情况。

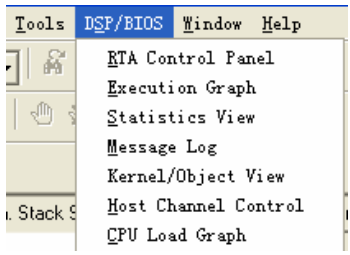


图 7.9　DSP/BIOS 工具

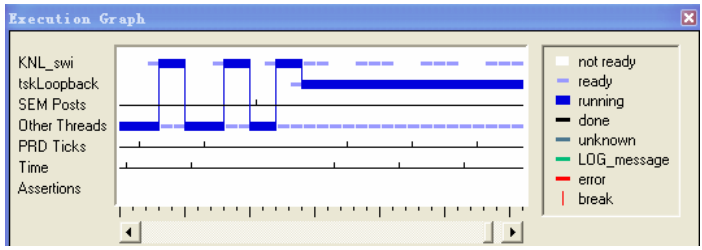


图 7.10　Execution Graph 窗口

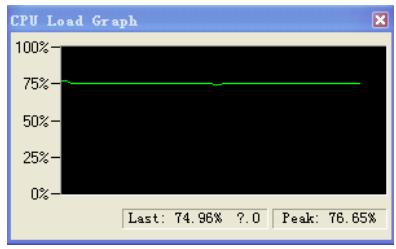


图 7.11　CPU Load Graph 窗口

注意：以上相关工具所使用的传递系统当前信息的函数都工作在 IDL 任务中（见图 7.12），具有最低的优先级，因此在系统满载的时候无法保证数据的实时更新。

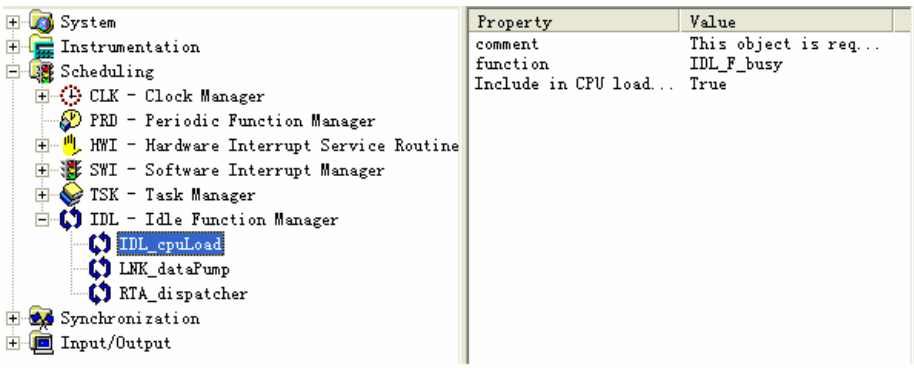


图 7.12　IDL Function Manager

3）CLK 模块

另外一个常用的评估程序性能的方法是使用 CLK（时间管理）模块，通过以下函数可以获得当前时间信息（高精度）：

```
currtime=CLK_gethtime();
```

上述函数将返回当前的处理器时间（不是时钟数，换算成实际指令时间时还需乘以系数 CLK_cpuCyclesPerHtime()，对于 DM642 来说该系数是 8）。需要注意，这里因为 int 数据类型的限制，currtime 在 24 s 后将会归零（720 MHz 处理器频率的条件下）。如果需要更长的测量时间，可以使用低精度的计时函数取代，如

```
currtime=CLK_getltime();
```

还有就是以上两个函数都不能在 main 函数中调用，因为此时 Timer 还没有准备好。

需要强调的是上面所介绍的各种工具都有自己的优点也有相应的局限之处，因此在实际的程序优化过程中往往需要多种工具一起使用，才能获得期望的效果。

7.4.2　程序优化

根据 TMS320C6000 Programmer's Guide 上的建议，标准的 DSP 开发流程如图 7.13 所示，是一个反复迭代的过程。因为篇幅所限，无法详细介绍 DSP 的体系结构及如何在此基础上写出高效的汇编代码，在这里主要想以示例来说明两种相对简单的 DSP 程序优化思路：编译器优化和从算法自身的角度进行优化。

1）编译器优化

DM642 基于 VLIW 结构，编译器对程序性能有很大的影响，因此通过调整编译器选项进行程序优化是相对简单但又极其重要的一步，这里以上面所提供的最原始直方图均衡化算法为例，说明编译选项对最终程序速度的影响。调整参数为 OptLevel（-o 选项），不同参数下的程序性能如表 7.1 所示。

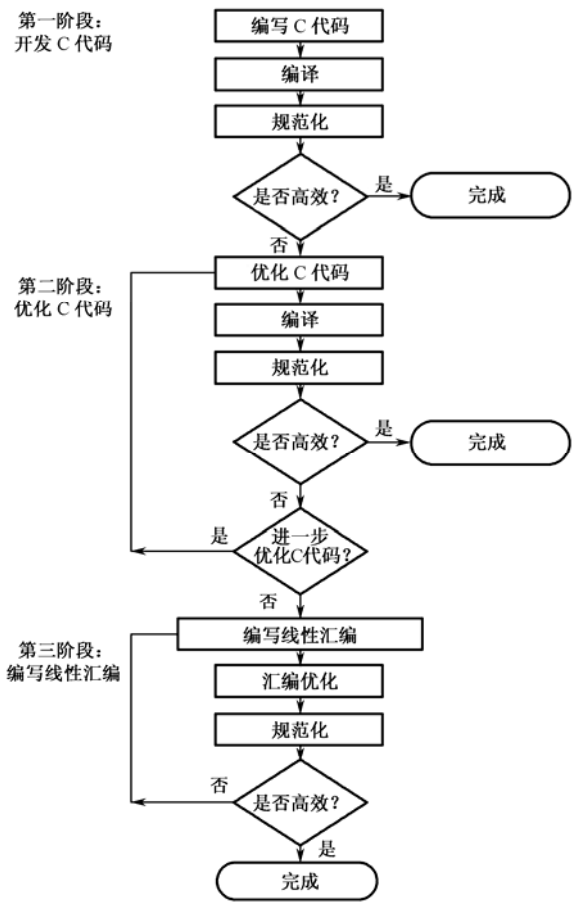


图 7.13　标准 DSP 程序优化流程

表 7.1　不同参数下的程序性能

	无　优　化	-o1(Local)	-o3(FIle)
帧率/(f/s)	15.2	25.0	30.0
CPU 占用率	98%	98%	61%

从表中可以发现，通过调整优化选项可以极大提高程序的运行效率，由无法实时（无优化，每秒 15 帧）到实时（o3 优化，61%占用率），可以说是一种最快捷的优化方法，但是这种优化并不是没有代价的，一方面文件级的优化改变了代码顺序关系，不利于调试；同时有一些优化选项，如 Program Level Optimization（-pm 选项）需要知道文件的依赖关系，使用时必须谨慎。

2）算法优化

除了使用编译器进行优化以外，从算法本身进行优化则是一种更为标准的做法。在优化算法之前，第一步也是最重要的一步是找出系统的瓶颈，只有找到占用系统资源最多的部分进行优化，才能做到事半功倍。回到这里讨论的直方图均衡化算法，通过使用 7.4.1 节介绍的仿真工具，可以发现整个函数大部分的运算时间都花费在第 22～30 行上，如下所示。

```
%022        for(i = 0; i < numLines; i ++) {
%023            for(j= 0;j<disLinePitch;j++)
%024            {
%025                currentPoint=*(disFrameBuf->frame.iFrm.y1 + i *
disLinePitch+j);
%026                if(currentPoint<0)
%027                    currentPoint=currentPoint+256;
%028                nk[currentPoint]=nk[currentPoint]+1;
%029            }
%030        }
```

造成这一点的主要原因是统计直方图时，使用了以下语句：

```
nk[currentPoint]=nk[currentPoint]+1;
```

因为编译器无法预测到下一个 currentPoint 是否会与当前的 currentPoint 相同，因此无法把 023-023 这样一个数据相当庞大（480×720）的循环进行软件流水化，这直接导致了系统运行的瓶颈。

处理上述问题的方法有很多，一个比较正统的解决方案是分治法，即同时对多个子图进行直方图统计，然后再进行综合，当然，这会增加代码的复杂性。另外一个思路是以精度换速度，考虑到这里的采样点很多，而统计直方图并不是每个点都需要，因此可以采用隔点采样的方法进行统计，如下所示。

```
……
%022        for(i = 0; i < numLines; i =i+2) {
%023            for(j= 0;j<disLinePitch;j=j+2)
%024            {
……
%029            }
```

```
        %030        }
        ……
```

经过这样以精度换时间的改进，CPU 的占用率会下降到 43%左右，而图像质量基本保持不变。以此为思路进一步扩展，可以先找出图中有意义的区间（如对比强烈的部分），然后仅以这部分的像素值进行均衡化，这样就推导出基于内容的图像增强方法。具体的实现可以参考相应文献，这里不再进一步叙述。

参 考 文 献

［1］R. C. Gonzalez, R. E. Woods. Digital Image Processing: Prentice Hall, 2007.

第 8 章　使用 MATLAB 开发 DSP 的图像处理算法

在第 7 章介绍了如何使用 Code Composer Studio 实现基本的图像增强算法，但很多时候，我们对算法实现的效率并不那么关心，更需要一种快速的开发方法，能够很快将我们原理性仿真的结果在硬件平台上实现与验证，以此为目的，直接通过 MATLAB 开发 DSP 程序无疑是一个很好的选择，这也是本章要讨论的问题——如何使用 MATLAB 开发视频图像处理算法。

本章主要包括以下内容：MATLAB 开发 DSP 程序的基本流程和 CCS Link 工具的使用方法，使用 Embedded MATLAB 搭建一个标准的 Simulink 模块，最后给出一个示例，使用 MATLAB 实现与第 7 章相同原理的图像增强算法。

8.1　MATLAB Link for Code Composer Studio

8.1.1　背景介绍

传统的 DSP 应用系统设计流程分为两部分：算法设计和产品实现。在算法设计部分完成理论研究与验证，使用专门的科学软件（如 MATLAB）进行仿真；在产品的实现阶段将开发设计阶段的算法用 C/C++或者汇编语言实现，在硬件平台（如 DSP）上进行调试优化。这个过程冗长且要求对硬件平台有相当程度的了解，这无疑增加了产品开发的周期和难度。

为了解决这个问题，出现了系统级设计方法的构想，系统级设计方法的核心是将算法设计和产品实现在统一的开发环境中进行,从而有效地将开发流程的两部分结合在一起。进行系统级设计需要一个统一的开发环境，该开发环境不仅可以对系统结构、算法进行描述，还能够对系统不同层次、不同组件和不同数据类型进行建模。MATLAB Link for Code Composer Studio 就是一种完成系统级设计的可行方案。

Mathworks 公司和 TI 公司联合开发的 MATLAB Link for Code Composer Studio（CCS Link），提供了 MATLAB 和 CCS 的接口，即把 MATLAB 和 TI CCS 及目标 DSP 连接起来，利用此工具，可以在 MATLAB 中通过 Simulink 建模的形式完成 DSP 程序的设计、开发与调试。CCS Link 具有以下特性（CCS Link 3.1）：

（1）仅使用 MATLAB 函数即可以自动完成 CCS 端的调试、数据传输和验证。

（2）在 MATLAB 和 DSP 之间使用 RTDX 实时传输数据。

（3）支持 XDS510/XDS560 仿真器，可以高速调试硬件 DSP 开发板。

（4）提供嵌入式对象，可以访问 C/C++变量和数据。

（5）对测试、验证和可视化 DSP 代码提供帮助。

目前，CCS Link 可以支持 CCS 能够识别的任何目标板，包括 TI 公司的 DSP，EVM 板

和用户自己开发的目标 DSP（C2000，C5000，C6000）板等。

8.1.2　安装配置

从 MATLAB 2006a 开始，CCS Link 已成为标准 MATLAB 组件的一部分。这里的仿真在 MATLAB 2007b 上进行，当前的 CCS Link 版本为 3.1。

CCS Link 安装完毕以后，可以通过以下 MATLAB 命令进行验证：

```
ccsboardinfo
```

上述命令后，MATLAB 将会输出当前 CCS Link 的配置状态，如果配置成功将会出现类似下面的信息：

```
Board  Board                        Proc   Processor   Processor
Num    Name                         Num    Name        Type
 0     DM642 EVM XDS560 Emulator    0      CPU_1       TMS320C64xx
```

这里我们可以看到 CCS Link 发现了一个 DSP 连接（DM642 EVM XDS560 Emulator），并成功地检测出其中的处理器类型为 TMS320C64xx，对应的 CPU 名称为 CPU_1。

此外，我们也可以在 Code Composer Studio 中检验 CCS Link 是否安装成功，正确安装条件下 CCS 的 Component manager 中应该有图 8.1 所示的信息。

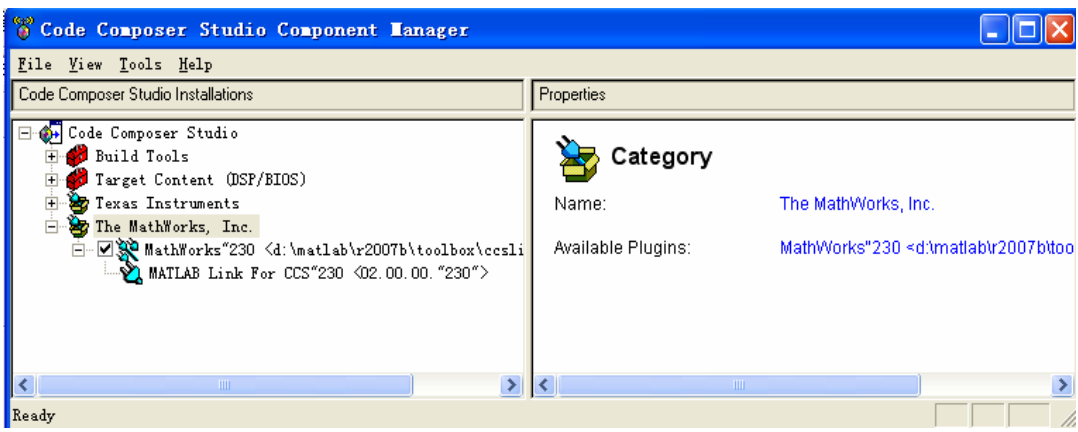


图 8.1　CCS Link 配置

8.2　示 例 程 序

下面以 MATLAB 提供的边缘检测（Edge Detection）演示程序为例，讲解使用 CCS Link 开发 DSP 应用的基本框架，该示例可在$InstallDir\toolbox\rtw\targets\tic6000\tic6000demos 目录下找到，如图 8.2 所示。

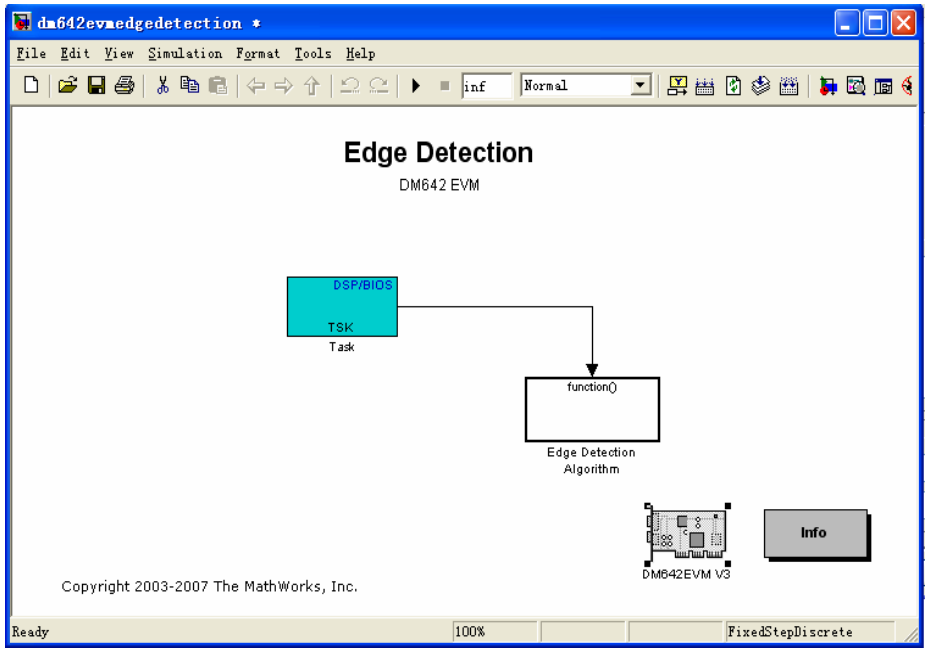


图 8.2　示例程序：边缘检测

这是为一个标准的 Simulink 模块，其中 DM642EVM V3 为当前使用的开发板型号，参数可以通过双击该模块进行设置，使用时需根据实际开发板的类型进行调整，如图 8.3 所示。

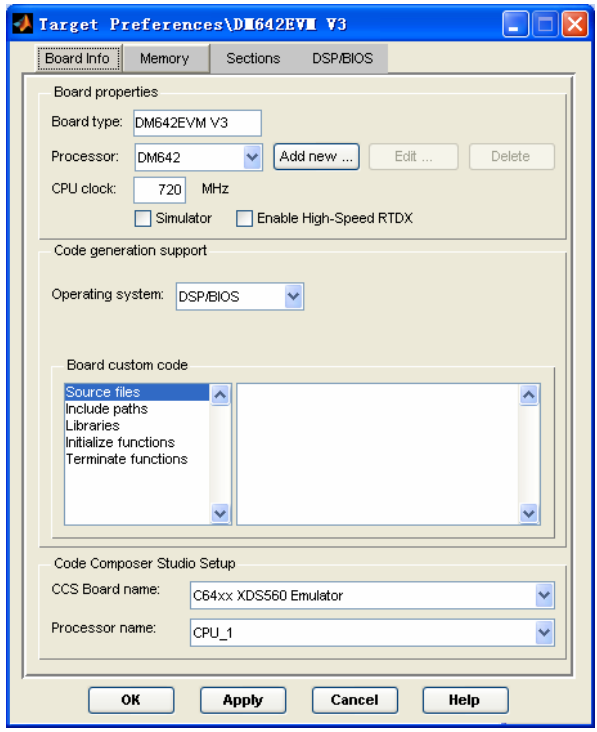


图 8.3　Simulink 中 EVM 开发板配置界面

这里使用了 XDS560 仿真器，在该配置界面中实现了传统 CCS 开发中 tcf 配置工具的大部分工作，如 CPU 的类型、频率设置、内存的分配（包括内部存储器和外部存储器）、程序存储位置设置等。

Task 模块声明任务类型（参考 DSP/BIOS 的介绍部分），参数如图 8.4 所示，注意这里同第 7 章一样，需要根据任务的需要设置优先级（这里为 1 级），同时调整 Stack 的位置（这里选择 SDRAM 存储器）与大小（这里为 4 096 B），以防止任务在运行时出现堆栈溢出的情况。

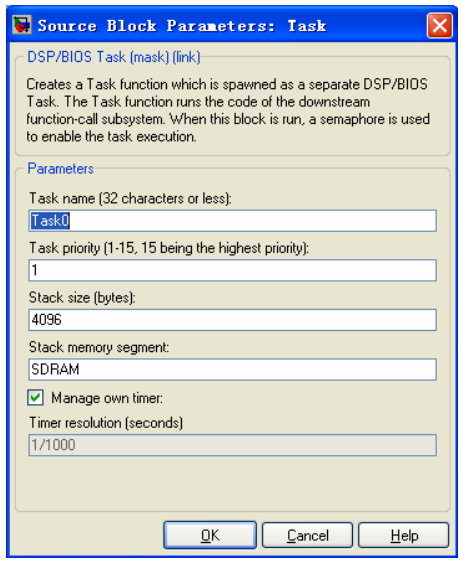


图 8.4　Task 设置

实际信号处理的算法在 Edge Detection Algorithm 块中实现，打开该模块，可以看到如图 8.5 所示的模块细节。

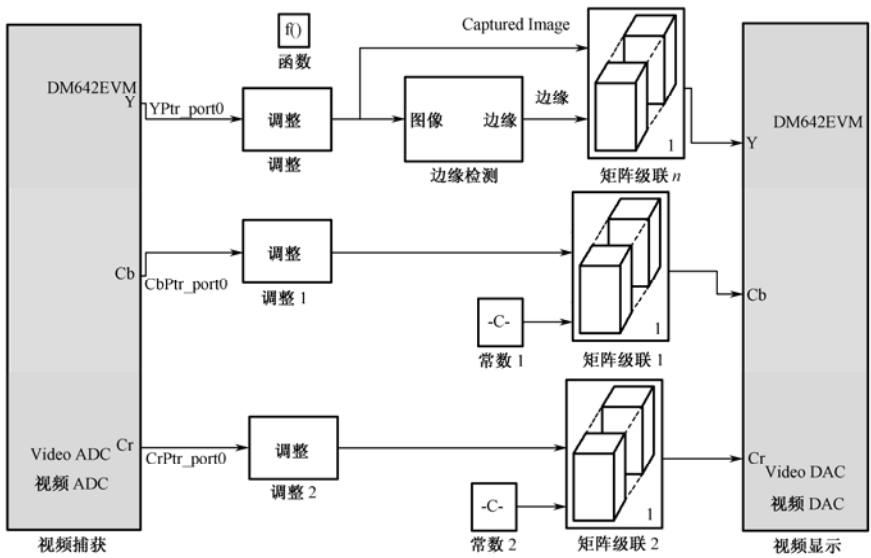


图 8.5　边缘检测算法

图 8.5 中各模块的含义介绍如下。

Matrix Concatenation模块：矩阵连接模块，由Simulink默认提供，其目的是在一个屏幕上同时显示经过缩放的原始图像与处理后的图像，如 图 8.11中将原始图像与增强后图像拼接在一起显示。

Resize 模块：对原始图像进行缩放，以便后续的 Matrix Concatenation 模块能够得到一个满足输出分辨率要求的图像。

Video Capture 模块（见图 8.6）与 Video Display 模块（见图 8.7）：视频捕捉与显示模块用于从输入设备（摄像头）中获取图像并将处理好的图像在输出设备（LCD 显示器）中显示。这里特别要注意需要调整 Video Capture 和 Video Display 上的相应芯片参数，以保证使用的编解码芯片和实际开发板保持一致，如这里 DM642 使用的是 TVP5146 芯片。

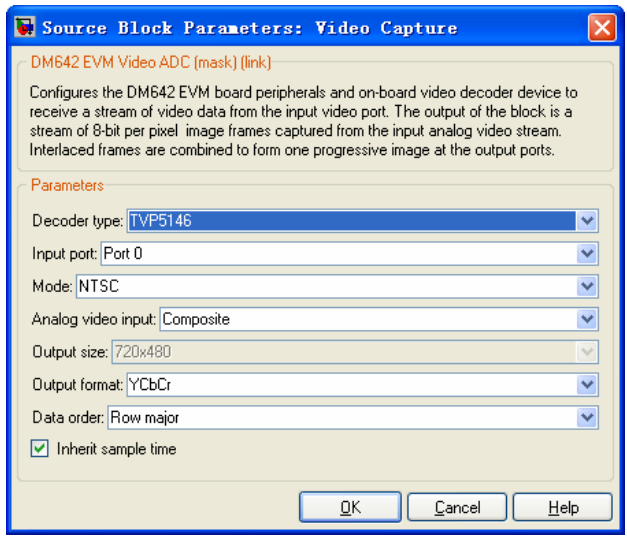


图 8.6　Video Capture 模块

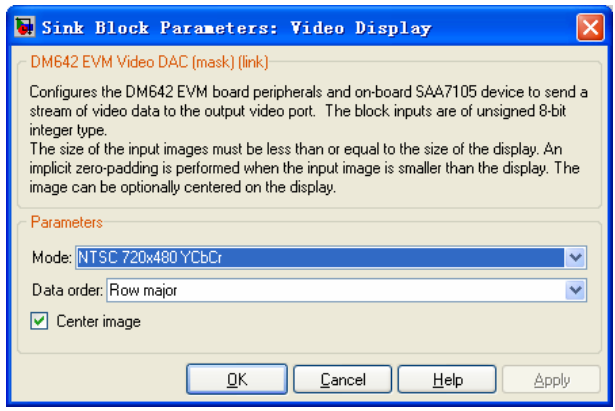


图 8.7　Video Display 模块

实际的边缘检测在 Edge Detection 模块中实现，进一步打开边缘检测模块，有如图 8.8 所示内容。

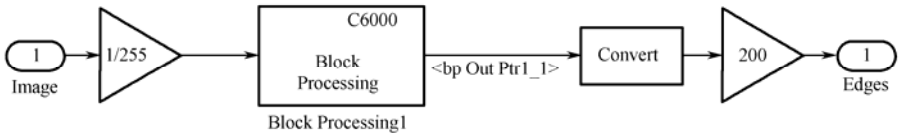


图 8.8　边缘检测模块

这里输入图像与输出分别以 Image 与 Edges 表示，其中因为输入图像为[0, 255]之间，因此在图像数据进入处理前需要进行归一化处理（1/255 模块），在输出时再通过标准化转化为显示器能显示的图像值。而主要的边缘检测算法在 Block Processing 模块中实现，这里直接使用了 Simulink 中 Video and Image Processing Blocks 的边缘检测模块，如图 8.9 所示。

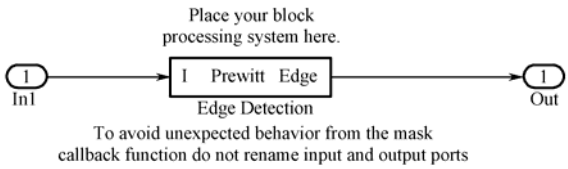


图 8.9　边缘检测实现

将上述 Simulink 工程进行增量编译以后，CCS Link 会完成 MATLAB 和 CCS 之间的连接工作，在 CCS 中新建一个工程（如这里是 dm642evmedgedetection.pjt，见图 8.10），并自动完成编译、加载与运行的工作。其中 dm642evmXXXXXX.c 即为 CCS link 根据上面的 Simulink 模块转化成的可在 DSP 上工作的 C 代码。

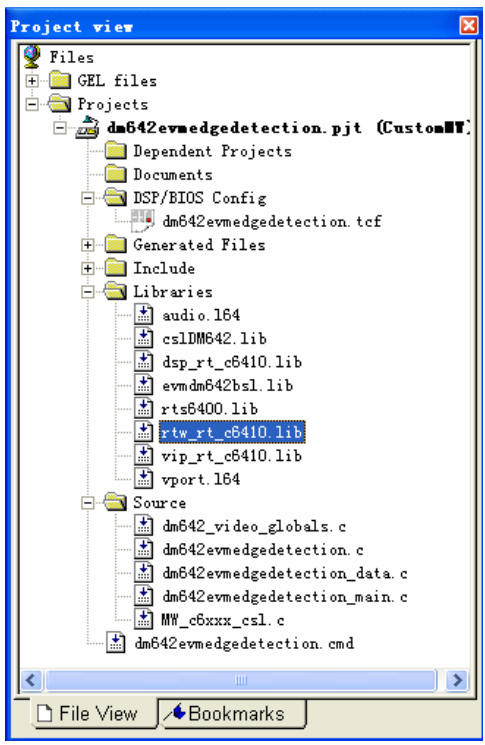


图 8.10　连接 CCS 后新建项目视图

如果硬件连接和各项设置正确，最终的结果如图 8.11 所示，注意这里左半部分为原始视频缩放的结果（见图 8.5 的 resize 块和 Matrix Concatenation 块），右半部分为计算得到的边缘。这里我们的测试场景为一个正常光照条件下的电脑照片，通过边缘检测模块处理后，可以很好地获得图像地边缘。

如果想开发自己的图像处理算法，也可以由此入手，对图 8.8 中的 Block Processing 模块进行修改。下面将介绍如何编写自己的 Simulink 模块，8.4 节将给出一个图像增强的示例。

图 8.11　边缘检测实际运行结果

8.3　使用 Embedded MATLAB 构造 Simulink 模块

正如在 8.2 节介绍的，可以通过编写 simulink 模块来实现新的算法，这些算法可以是已有的（如 Video and Image Processing Blocks 中的相关视频处理和图像处理模块），也可以是自己编写的模块。Simulink 提供了一系列通过 MATLAB 函数自定义模块的方法（MATLAB Function Blocks），其中常用的有以下几种。

1）Fcn Block

通过输入表达式创建单输入单输出模块（single-input，single-output(SISO) block）。这里的表达式可以直接调用部分 MATLAB 函数，但必须保证该函数为单输入单输出，如 sin（u.*u）（u 为默认的输入信号），Fcn Block 配置界面如图 8.12 所示。

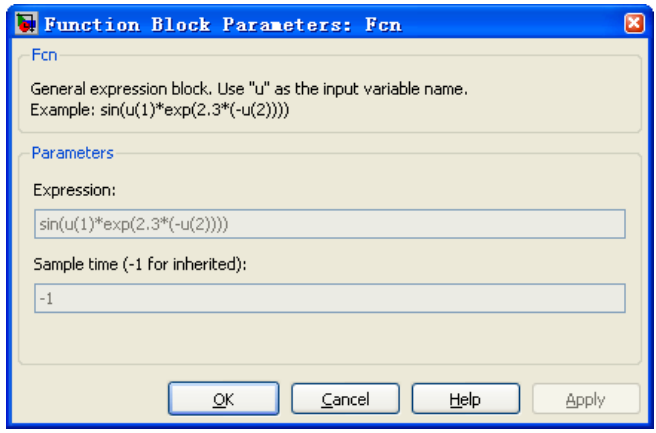


图 8.12　Fcn Block 配置界面

2）MATLAB Fcn Block

允许使用 MATLAB 函数定义一个固定输出维度的 Block。与 Fcn Block 不同，这里可以使用自定义的 MATLAB 函数，但因为在每次调用时都需要 MATLAB 进行解析，因此速度比 Fcn Block 要慢，MATLAB Fcn Block 配置界面如图 8.13 所示。

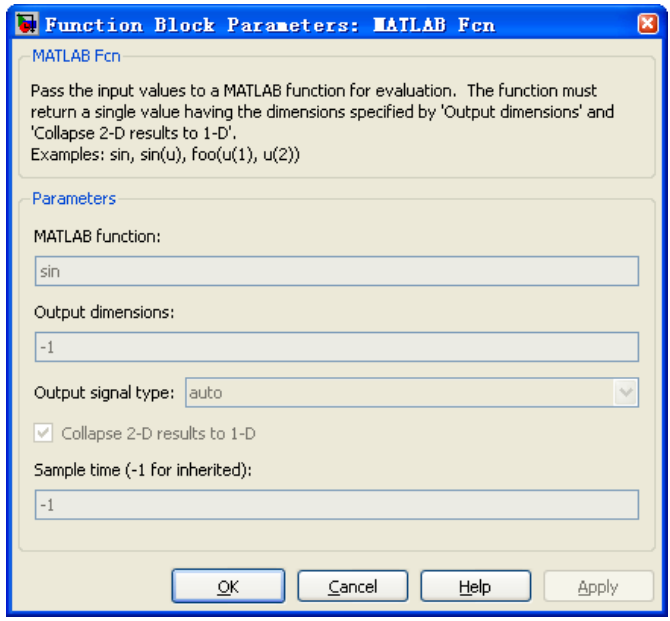


图 8.13　MATLAB Fcn Block 配置界面

3）Embedded MATLAB Function

通过使用 Embedded MATLAB 语言描述一个多输入、多输出的模块，这种方法具有最好的普适性，也是在嵌入式平台上效率最高的方法。

本节主要讲述如何通过使用 Embedded MATLAB 编写一个自己的 Simulink 模块。

8.3.1　Embedded MATLAB 简介

简单地说，Embedded MATLAB 可以看做是 MATLAB 语言的一个子集，它专门为嵌入式算法设计，与标准的 MATLAB 开发流程不同，通过 Embedded MATLAB 开发的模块并无法直接通过解释器运行，而需要经过一个编译的过程，最终得到目标平台 C 语言，之后调用目标平台的专有编译器，得到可在嵌入式平台上运行的可执行代码。在 MATLAB 端的标准开发流程如图 8.14 所示。

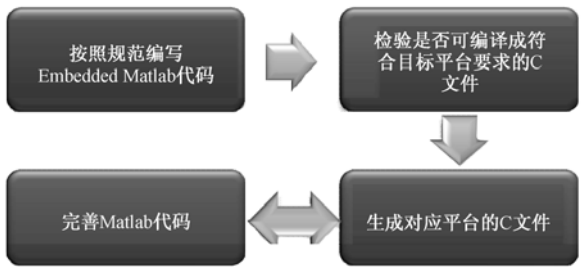


图 8.14　Embedded MATLAB 端的标准开发流程

因为嵌入式平台的特殊性，Embedded MATLAB 对标准解释性语言进行了修改，主要是增加了一系列限制，具体如下所述。

1）数据类型限制

出于效率的考虑，在 MATLAB 语言转换成 C 语言的工程中使用静态转化，因此在使用数据之前需要对所有变量的数据类型与纬度进行声明，不允许数据类型的动态转化，如下所示：

```
x = 2.75; % OK
y = [1 2; 3 4]; % OK
x = int16(x); % ERROR: cannot recast x
y = [1 2 3; 4 5 6]; %ERROR: cannot resize y
```

在上面代码中 x 被定义为浮点型，因此不能通过 int16 函数强制转化为 int 类型；而 y 的维度也被固定为 2×2，不能通过赋值语句加以改变。

Embedded MATLAB 支持表 8.1 所示的数据类型。

表 8.1　Embedded MATLAB 支持的数据类型

char	字符，字符串
complex	复数
double	双精度浮点数
int8, int16, int32	有符号整型
logical	布尔值（true/false）
single	单精度浮点数
struct	结构体
uint8, uint16, uint32	无符号整数

这里需要注意的是 Embedded MATLAB 不支持元数组与动态变量。

2）内存分配限制

出于速度和安全的考虑，Embedded MATLAB 不支持动态内存分配，在一般 MATLAB 中常用一些矩阵初始化方法，如

```
M(i:j) =1
```

在 i，j 发生变化时将无法工作，与之对应，Embedded MATLAB 中一般使用如下的矩阵初始化方法：

```
M = ones(10,10);
for i=1:10
   for j = i:10
       M(i,j) = 2 * M(i,j);
   end
end
```

即先通过 ones(M, N)（或 zeros(M,N)）对变量维度与类型进行定义，然后使用矩阵下标进行初始化。

3）速度与内存限制

因为 Embedded MATLAB 程序在转化为实际可执行文件之前需要有一个编译成 C 语言的过程，因此效率相对于 C 语言直接实现的程序会差很多；同时嵌入式设备一般是内存受限的，因而在编写的时候必须对变量使用循环的展开等诸多因数加以考虑，这在一定程度上增加了程序开发的复杂性。

尽管有诸多限制，Embedded MATLAB 仍然不失为一门相当易于使用的语言；更重要的是 MATLAB 中有大量的现有函数可以直接调用，其中包含了 270 个 MATLAB 函数及超过 100 个 Fixed-Point Toolbox 函数。从语言角度来看这很大程度上降低了开发的难度，如在 8.4 节示例 8-1 中使用的统计直方图函数 hist（008 行）。

8.3.2　如何使用 Embedded MATLAB 开发 Simulink Blocks

Simulink 为创建自定义的 Block 提供了一系列接口，如图 8.15 所示。

打开其中的 Embedded MATLAB Function 模块，即可看到 Embedded MATLAB Editor 编辑器及一个最简单的例程，如下所示：

```
function y = fcn(u)
% This block supports the Embedded MATLAB subset.
% See the help menu for details.
y = u;
```

这里 Embedded MATLAB 的定义方式与传统 MATLAB 函数文件定义相同，都通过 Function 关键词加以声明，主要区别在于该函数为封装在 Simulink 模块中。下一节将从该例程出发，给出一个使用 Embedded MATLAB 实现的图像增强算法示例。

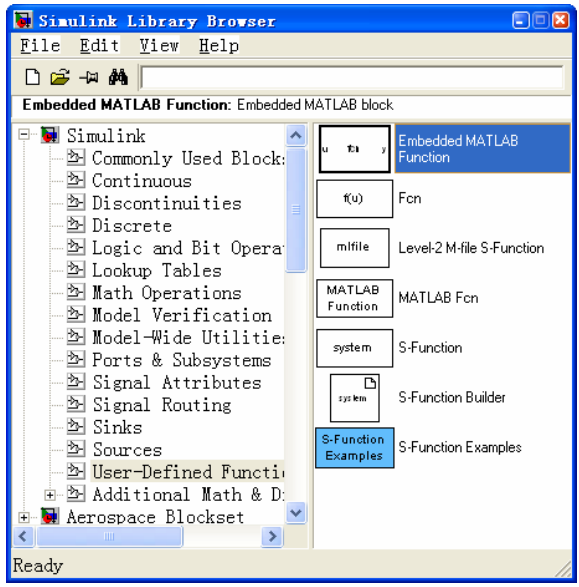


图 8.15　Embedded MATLAB Block 提供的接口

8.4　使用 MATLAB 开发的视频图像增强算法

整个 Simulink 的框架如同 8.2 节（见图 8.2，图 8.5），这里所需要修改的是将图 8.8Edge Detection Block 模块以 Embedded MATLAB 所写的 Image Enhacement 模块替代，如图 8.16 所示。

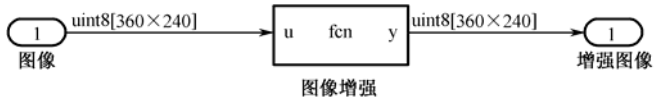


图 8.16　基于 Simulink 的图像增强模块

其中 Image Enhancement 最终的实现代码见示例 8-1。

示例 8-1　使用 Embedded MATLAB 实现图像增强算法。

```
%001 function y = fcn(u)
%002 L=256;
%003 sk=zeros(1,L);
%004 Length=360;Width=240;
%005 y=uint8(zeros(Length,Width));
%006 n=Length*Width;
%007 B=reshape(u,n,1);
%008 nk=histc(B,[0:L-1]);
%009 sk(1)=nk(1);
%010 for k=2:L
%011    sk(k)=sk(k-1)+nk(k);
```

```
%012 end
%013 sk=floor(sk*256/n);
%014 for i=1:Length
%015    for j=1:Width
%016       y(i,j)=uint8(sk(u(i,j)+1));
%017    end
%018 end
```

在上述代码中需要注意以下内容。

001 行：Embedded MATLAB 中通常使用 y 表示输出信号，u 表示输入信号。

005 行：MATLAB 中图像信号以 uint8 格式表示，由 Embedded MATLAB 的要求在使用数据前对数据的类型需要进行声明与初始化。

007 行，008 行：直接使用 MATLAB 函数 histc 进行直方图统计。

016 行：保证数据类型的一致性。

最终的仿真结果在 LCD 显示器上显示为图 8.17 所示。这里我们选择的测试源为昏暗条件下的手表图像，与上面的边缘检测示例一样，对输出的图像进行矩阵变换和拼接，其中左图为原始图像，右图为经图像增强后得到的图像。

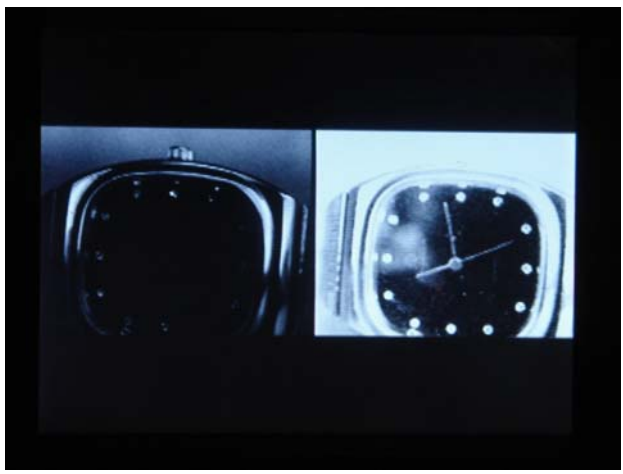


图 8.17　DSP 处理结果对比

在仿真结果中可以发现，由 MATLAB CCS Link 生成的代码效率较直接实现的代码低，画面有明显丢帧的现象出现。虽然如此，我们已经可以对该算法进行评估，能够发现使用直方图均衡化进行图像增强存在的一个重要问题：使用了全局信息而忽略了局部的信息，直接结果便是这里表面指针由之前的昏暗不可见变得清晰，但原图中一些其他清晰的细节信息则变得不可见。同时示例 8-1 的代码复杂程度远小于使用 C 语言实现的代码，且 MATLAB 语言的易用性(动态内存指定大量可用的函数模块)使得代码调试时间大为减少，因此 MATLAB 与 CCS Link 组成了一个针对行视频处理算法的验证性开发的良好平台。

参 考 文 献

［1］张众．MATLAB 开发 DSP 的图像处理算法．多媒体信号处理实验室技术报告．清华大学电子工程系，2008.

［2］Matlab Link for Code composer studio development Tools，Mathworks.